コンピュータ理工学部特別研究 II A・B

固有値計算手法を用いたランキング計算機構の実装とその性能チューニング

Implementation and performace thuning of the ranking calculation subsystem using eigenvalue caluclation

秋山特研

尾﨑 拓也

平成 24 年 3 月 28 日

概要

我々の研究グループでは，検索サービスおよびソーシャルコミュニケーションの双方の利点を同時に活用した，ソーシャルサーチシステムを提案している．本研究では，提案システムのランキング機構で必要となる大規模行列の固有値計算機能を実装し，性能チューニングにより計算の高速化を目指す．固有値計算には並列計算ライブラリ SLEPc を用いた．SLEPc は既存の数値計算ライブラリを用いて固有値計算手法を実装しており，大幅に計算時間を短縮できる．実際に SLEPc を用いて計算を行ったところ，70 万 $\times$ 70 万 程度の遷移確率行列を 48 プロセッサを用いることで，20.29 秒で計算できた．ここで，Ubuntu および CentOS の環境を用いて計算したところ，両者の性能には予測していたよりも大きな差が認められた．様々な調査の結果，性能差の原因は BLAS のサブルーチンである dgemm と dgemv という行列・ベクトル計算ルーチンであることが分かった．そして，GotoBLAS を用いて最適化をすることで，より少ない 16 プロセッサでの計算でも 19.51 秒で計算できることを確認した．

Abstract

Our research group proposes social search system that takes advantages of both the search service and the social communication. In this study, we implement the large-scale eigenvalue calculation routines required by the ranking subsystem of the proposed system and improve the calculation performance by tuning system parameters. We used SLEPc to calculate the eigenvalues. SLEPc uses existing libraries such as OpenMPI and PETSc, and reduces the computation time substantially. We could complete the eigenvalue calculation of markov model matrix with about $700,000 \times 700,000$ in 20.29 seconds by using 48 processors. When we compared the results of the two evaluation environments that using Ubuntu and CentOS, there were unexpected difference of the performance. As a result of the various investigation, it was caused by dgemm and dgemv that were BLAS subroutines and calculating matrix and vector multiplication. After the optimization by the GotoBLAS, we could complete the calculation in 19.51 seconds using 16 processors which were less than the previous case.

目 次

1 はじめに **4**

2 ソーシャルサーチシステムとランキング **4**
2.1 PageRank について . . . 5
2.2 固有値計算手法を用いたランキング . . . 5
2.3 固有値計算について . . . 7
2.4 SLEPc . . . 9

3 ランキング計算手法の事前評価 (Ubuntu, DELL) **9**
3.1 評価指標 . . . 10
3.2 単一ノードでの性能評価 . . . 12
3.2.1 単一ノードでのラプラシアン行列の性能評価 . . . 12
3.2.2 単一ノードでの遷移確率行列の性能評価 . . . 13
3.3 複数ノードでの性能評価 . . . 14
3.3.1 複数ノードでのラプラシアン行列の性能評価 . . . 14
3.3.2 複数ノードでの遷移確率行列の性能評価 . . . 15
3.4 事前評価に伴う考察とその対策 . . . 15
3.4.1 Hyper-Threading による影響 . . . 15
3.4.2 –bind-to-core オプション . . . 17
3.4.3 通信によるオーバヘッドの影響 . . . 18

4 ランキング計算手法の評価 (CentOS, HP) と異種環境間での性能比較 **19**
4.1 単一ノードでの性能評価と比較 . . . 19
4.1.1 単一ノードでのラプラシアン行列の性能評価と比較 . . . 19
4.1.2 単一ノードでの遷移確率行列の性能評価と比較 . . . 21
4.2 複数ノードでの性能評価と比較 . . . 21
4.2.1 複数ノードでのラプラシアン行列の性能評価と比較 . . . 21
4.2.2 複数ノードでの遷移確率行列の性能評価の比較 . . . 23
4.3 social における通信によるオーバヘッドの影響 . . . 23
4.4 考察 . . . 25

5 異種環境間での性能差の原因調査 **25**
5.1 使用するライブラリのバージョンについて . . . 26
5.1.1 関数の呼び出し回数の比較 . . . 26
5.1.2 使用するライブラリのバージョンの変更 . . . 26
5.2 仮想マシンを用いた調査 . . . 27
5.3 Ubuntu と CentOS の性能差の原因調査 . . . 29
5.3.1 sysctl を用いた調査 . . . 29
5.3.2 strace と htop を用いた調査 . . . 29

5.3.3 -log_summary オプションを用いた調査 30
5.3.4 gprof を用いた調査 31

6 行列・ベクトル計算の最適化 32
6.1 ATLAS, GotoBLAS とは 32
6.2 仮想マシンを用いた ATLAS, GotoBLAS の性能評価 33
6.3 ATLAS, GotoBLAS に関する考察 33
6.4 agile, social への ATLAS, GotoBLAS の適用 34

7 まとめと今後の課題 34

1 はじめに

近年，スマートフォンの普及や，Twitter, Facebook といったコミュニケーションサービスの登場により，多くの人がインターネットを利用して，様々な情報をやり取りしている．それに伴い，情報を検索する際に大量の Web ページが表示され，自分の本当に知りたい情報を得ることが困難になっている．情報を検索する手段として，Google や Yahoo!などの検索サービスを用いた高速で網羅的な情報が得られるものと，Twitter や mixi, Facebook などを介した人とのコミュニケーションからもたらされる質の高い情報が得られるものの 2 つがある．我々の研究グループでは，検索サービスおよびソーシャルコミュニケーションの双方の利点を同時に活用した，ソーシャルサーチシステムを提案している．提案システムでは，Web ページを閲覧しているユーザのネットワークをリアルタイムに構築することで閲覧者の量と質に基づいたランキング機構，ならびに，ページを通した閲覧者とのリアルタイムなコミュニケーション機能により検索とコミュニケーションの融合を目指す．提案システムにより，閲覧者が知りたい情報と知識を豊富に持つユーザと瞬時にコミュニケーションを図ることが可能になり，さらに質の高い情報を獲得することができる．提案システムでは PageRank アルゴリズムを拡張し，各ページ間のリンクに対して閲覧者のアクセス数とアクセス時間を考慮した重み付けを付与するランキング手法を用いる．リアルタイムに閲覧者の興味を反映したランキングを行うには大規模なランキング計算を短時間で行う必要がある．本研究では，ランキング時に行われる大規模行列の固有値計算の実装を行い，性能チューニングにより，計算の高速化を目指す．以下，2 章ではソーシャルサーチシステムとランキング計算について述べ，3 章ではランキング計算手法の事前評価について述べる．4 章では異種環境間での性能比較について述べ，5 章では異種環境間での性能差の原因調査について述べる．6 章では行列・ベクトル計算の最適化について述べ，最後に 7 章では本研究のまとめと今後の課題について述べる．

2 ソーシャルサーチシステムとランキング

ソーシャルサーチシステムは，図 1 のように，コミュニケーション機構，クローリング機構，ランキング機構，全文検索機構の 4 つに分けられる．コミュニケーション機構ではユーザの閲覧履歴の収集やユーザ間の関係を抽出し，クローリング機構ではページ間のリンク構造を抽出する．ランキング機構では，コミュニケーション機構とクローリング機構が収集した情報を，PageRank[1] に基づいて隣接行列の作成し，ランク値計算を行い，全文検索機構によって，より質の高い検索結果を返す．本研究では，ランキング機構において，固有値計算を用いたランキング計算手法を実装し，その性能チューニング方法について述べる．調査を行った．以下では，2.1 節で PageRank について述べ，2.2 節で固有値計算手法を用いたランキング計算手法について述べる．2.3 節では固有値の計算手法について述べ，2.4 節では固有値計算ライブラリについて述べる．

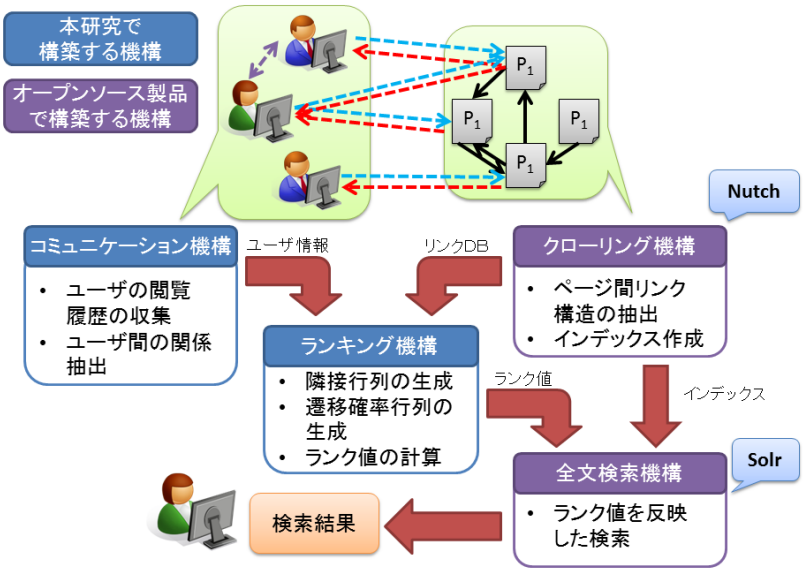

図 1: ソーシャルサーチシステムの構成図

2.1 PageRank について

PageRank は，文献 [1] で提案されているページの重要度の自動判定技術であり，Web ページ [2] にその概要が紹介されている．PageRank は，”多くの良質なページからリンクされているページはやはり良質なページである” という再帰的な関係をもとに，すべてのページの重要度を判定する．PageRank 値は正の実数値で与えられ，各 Web ページだけでなく，リンクにも与えられる．各 Web ページがもつインリンクの PageRank 値の合計値が各 Web ページの PageRank 値である．つまり，多くインリンクを持つ Web ページ，および重要な Web ページからのインリンクをもつ Web ページを，重要な Web ページであると判断する．

図 2 は Page らが PageRank 値の考え方を説明する際に用いた図である [1]．”リンク元の Web ページの PageRank 値”が 100 で”リンク元の Web ページのアウトリンク総数”が 2 本である場合，1 本あたりのアウトリンクがリンク先の Web ページに与え PageRank 値は 50 である．また，PageRank アルゴリズムはランダムサーファーモデルを用いている．ランダムサーファーモデルとは，ランダムにハイパーリンクをたどっていく多数の Web サーファーが Web ページの遷移を無限会繰り返して定常状態に達したときに，ある Web ページを閲覧している割合をその Web ページの相対的な重要度とみなすモデルである [1]．

2.2 固有値計算手法を用いたランキング

Web のようなハイパーリンク構造をランキングに反映させるためには，ハイパーリンク構造を計算機上でモデル化し，数値化する必要がある．ハイパーリンク構造をグラフ理論の応用と見た場合，線形代数の考え方に帰着されることが多く，PageRank も同様である．基本的にリンク関係を行列で表すことが多く，あるページ i から別のページ j へリンクが張られている場合にはその成分を 1 とし，そうでない場合を 0 とし

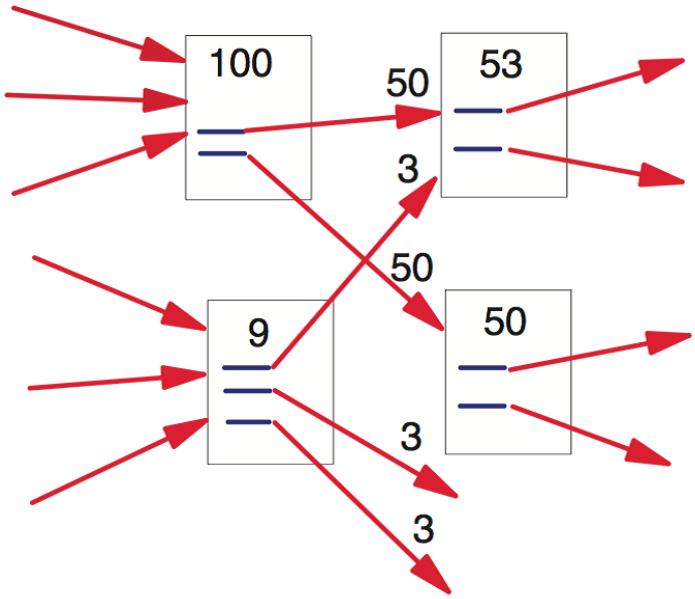

図 2: PageRank の考え方 文献 [1] から引用

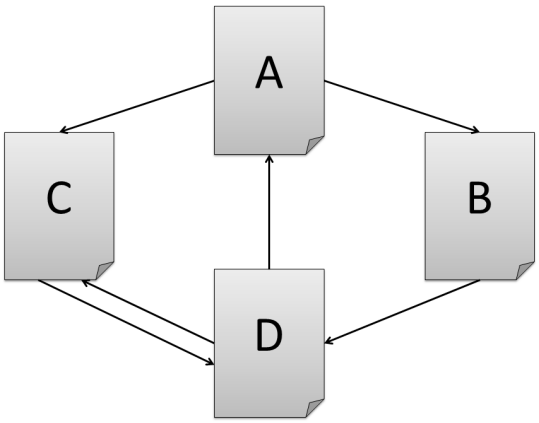

図 3: Web サイトのリンク構造図

て，行列を生成する．図 3 を例にして説明すると，ページ A はページ B とページ C にリンクをしているため，行列の (2,1) と (3,1) に 1 がたつ．同様に，ページ B, C, D のリンクを調べリンクのあるところに 1 をた

てると，下のような行列になる．

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

もしリンク数が N とすると，行列は $N \times N$ の正方行列になる．これは，グラフ理論で”隣接行列”と呼ばれる行列に相当する．PageRank で使用する行列は，生成した隣接行列の列ベクトルの総和が 1 になるようにそれぞれのリンク数で割ったものである．すなわち，図 3 を例にして説明すると，求めた隣接行列の列成分をリンク数で割る．1 列目はリンク B, C の 2 つなので各成分を 2 で割り，行列の (2,1), (3,1) に $\frac{1}{2}$ をたてる．同様に，他の列成分もリンク数で割ると下のような行列になる．

$$\begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

この行列は，”遷移確率行列”とよばれる．PageRank はこうして求めた遷移確率行列の固有値計算を行い，ランク値を求める．遷移確率行列を P，各ページのランク値を X とすると，定常状態におけるランク値は $PX = X$ を満たす．そのため，ランク値は固有値 1 に対する固有ベクトルとなる．

2.3 固有値計算について

固有値計算は定常波の解析から熱方程式，さらには量子力学など，広い分野で利用されており，これまでに様々な方法が提案されている．

固有値計算の古典的な方法として LR 法や QR 法がある．LR 法は行列が正方行列かつ対称行列である場合に計算することが可能であり，行列 A を対角要素がすべて 1 であるような左下三角行列 L と右上三角行列 R に分解をして計算する手法である．分解はガウスの消去法を用いることにより分解可能である．LR 法の計算手順は以下の通りである．

LR 法の計算方法

1. LR 分解を用いて左下三角行列 L と右上三角行列 R を生成する．
   $A_1 = L_1 R_1$
2. R_1,L_1 を逆にかけたものを A_2 とする．
   $A_2 = R_1 L_1$
3. 1,2 を繰り返し行う．
   $A_k = L_k R_k$
   $A_{k+1} = R_k L_k$

この計算を $k \to \infty$ 回行うことにより，A_{k+1} は三角行列に収束をし，A の固有値となる．

QR 法は，LR と違い非対称行列の場合にも計算が可能であり，行列 A をユニタリ行列 Q と右上三角行列 R に分解をして計算する手法である．ユニタリ行列とは，$A^*A = AA^* = E$ を満たすような行列のことである．ここで A^* は複素数共役転置行列，E は単位行列のことである．分解方法は，グラム・シュミット直行化に基づいて行う．

グラム・シュミット直行化法の計算方法

1. 行列の列成分に着目し，行列 A の 1 列目を a_1 とすると $a_1 = b_1$ を基底に設定し，正規化[a]を行い q_1 を生成する．
2. 行列 A の二列目 a_2 を用いて q_1 と直行な基底 $b2$ を生成する．
3. b_2 を正規化して q_2 を生成する．
4. これを繰り返し行い，q_1〜q_n を集めたものがユニタリ行列 Q となる．
5. 右上三角行列 R は $r_{ij} = (q_i, a_j)(i \neq j)$ または $r_{ij} = |b_i|(i = j)$ である．

[a]正規化とは大きさが 1 であるベクトルのこと．

QR 法の計算方法

1. QR 分解を用いてユニタリ行列 Q と右上三角行列 R を生成する．
   $A_1 = Q_1 R_1$
2. Q_1,L_1 を逆にかけたものを A_2 とする．
   $A_2 = Q_1 L_1$
3. 1,2 を繰り返し行う．
   $A_k = L_k R_k$
   $A_{k+1} = R_k L_k$ この計算を $k \to \infty$ 回行うことにより，A_{k+1} は三角行列に収束をし，A の固有値となる．

古典的な手法では，計算に大きな手間を要するため，固有値が等しく，元の行列サイズより小さい行列に変換することで計算時間を削減する”射影法”に基づく手法が提案されている．また，行列を対称行列の場合は三重対角行列に，非対称行列の場合はヘッセンベルグ行列に変換を行うことにより，計算時間の削減をすることができる．射影法を用いて計算を行っている Arnoldi 法，Krylov-Schur 法，Lanzcos 法などは，グラム・シュミットの手法をベースにした直交化のフェーズと，マルチシフト QR 法による固有値計算および Schur 分解のフェーズからなる手法で，分解フェーズでブロックを並べ替えることで最大固有値，最小固有値など，欲しい固有値および固有ベクトルの組から求められる手法である．そのため，最大固有値と対応する固有ベクトルのみが必要な場合では，大幅に計算時間を削減できる可能性がある．

2.4 SLEPc

SLEPc(Scalable Librart for Eigencalue Problem Computations) [4] とは，固有値計算問題のためのスケールライブラリである．SLEPc では，PETSc [5], BLAS, LAPACK などの既存の数値計算ライブラリを用いて固有値計算手法を実装しており，大幅な計算時間の短縮が期待できる．SLEPc が用いている PETSc という数値計算ライブラリは，Open MPI を用いてベクトルや行列の演算を複数プロセスで並列処理する機能を備えている．PETSc は C 言語で記述されており，高速な演算が期待できる．また，Open MPI はプロセス間通信方式として，共有メモリ，InfiniBand, TCP/IP など，さまざまな方式をサポートしており，内部で自動的に切り替えてくれるため，準備できる環境に応じて最適な並列化が用意に実現できる．SLEPc では固有値計算手法として，射影法を用いた Arnoldi, Krylov-Schur, Lanczos が提供されている．

3 ランキング計算手法の事前評価 (Ubuntu, DELL)

まず，SLEPc において本研究が目指すスケーラビリティを実現するためにどの程度のリソースを要するのかを調査する必要がある．そこで，ランキング計算手法の事前調査として，評価環境 agile(表 1) に SLEPc をインストールし，Arnoldi, Krylov-Schur, Lanczos を用いた固有値計算を行った．

表 1: 評価環境 agile

計算機	DELL PowerEdge R410 ×2
CPU	Intel(R) Xeon(R) L5520 (2.26GHz, 8MB キャッシュ, 5.86 GT/s QPI) ×2 (コア数 4)
メモリ	32GB (4GB×8/2R/1066MHz/DDR3 RDIMM)
ディスク	RAID6 (PERC6i), 600GB, 15,000RPM SAS
ネットワーク	Broadcom NetXtreme II BCM5716 1000Base-T PCI Express
スイッチ	Cisco Catalyst 2960G-8TC-L
OS	Ubuntu Linux 10.04
ライブラリ	PETSc 3.0.0, SLEPc 3.0.0, LAPACK 3.2.1, BLAS 1.2, OpenMPI 1.4.1

性能評価では，対称行列の例としてラプラシアン行列を，非対称行列の例として要素間をランダムに遷移するモデルを想定して生成した遷移確率行列を対象として固有値計算を行った．本実験で用いたラプラシアン行列は，図 4(a) に示したような行列で，対角成分が 2, その両脇の要素が -1 の行列とした．また，遷移確率行列は，三角格子の上をランダムに移動することを想定して生成した図 4(b) のような行列とした．本実験ではまず単一ノード内での並列処理性能について評価を行い，その後，TCP/IP を用いる複数ノードでの評価を行う．

$$\begin{bmatrix} -2 & 1 & & & & 0 \\ 1 & -2 & 1 & & & \\ & 1 & \ddots & & & \\ & & & \ddots & 1 & \\ & & & 1 & -2 & 1 \\ 0 & & & & 1 & -2 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ 0.5 & 0 & 1 & 0 & 0.5 & 0 \\ 0 & 0.25 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 & 0.5 & 1 \\ 0 & 0.25 & 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0 \end{bmatrix}$$

(a) ラプラシアン行列　　(b) 遷移確率行列

図 4: 対象とする行列

PETSc では，`-log_summary` というオプションを用いることで，プログラムの実行時間や事前設定した箇所の呼び出し回数とその実行時間を得ることができる．以下の実験では，`-log_summary` の機能を用いて性能計測を行った．また，SLEPc では，求める固有値の許容誤差を指定することで，計測結果が指定した誤差内に収まるまで反復計算を行う．以下の評価実験では許容誤差を 10^{-7} として測定を行った．

3.1 評価指標

評価実験では以下の指標を用いて性能評価を行う．

処理時間

本研究では，行列計算を開始してから終了するまでの時間を処理時間とし，`-log_summary` オプションにより得られるログを用いて処理時間を計測している．そのため，OpenMPI におけるローカルなプロセスの生成時間，ssh による遠隔プロセス生成時間などは含まれていない．

性能向上率 (speedup)

単一プロセッサで実行した処理時間に対する複数プロセッサ処理時の性能向上率を示す．単一プロセッサでの処理時間を T_1, n プロセッサでの処理時間を T_n としたとき，プロセッサ数を変化させたときの性能向上率 speedup は

$$speedup(n) = \frac{T_1}{T_n}$$

で表される．

並列化率

プログラム中での並列化可能部分の割合を表したものである．後述のアムダールの法則により，この値が低いと使用するプロセッサ数を増やしても，プロセッサ数に応じた性能を得ることができない，以下では，アムダールの法則とその並列化への適用について述べ，評価実験での並列化率の計算手順を示す．

- **アムダールの法則とは**
  アムダールの法則とは，プログラムの一部を改良したときに全体として期待できる性能向上の程度を知るための法則である．また，複数プロセッサを使ったときの論理上の性能向上率を予測するにも用いられる．

- **並列化へのアムダールの法則の適用**
  プログラム中で完全に並列化可能な部分の実行時間の割合 (並列化率) を R($0 \leq$ R ≤ 1) としたとき，並列化できない順次実行部分の割合は $1 - R$ となる．プロセッサ数 1 のときの実行時間を T_1 とすると，n 個のプロセッサを使用したときの並列化可能部分の実行時間は RT_1/n，順次実行部分の実行時間は $(1 - R)T_1$ となる．これよりプロセッサ数 n のときの実行時間を T_n とすると，

  $$T_n = (\text{並列化可能部分}) + (\text{順次実行部分}) = RT_1/n + (1 - R)T_1$$

  よって，性能向上率 speedup は，

  $$speedup(n) = \frac{T_1}{T_n} = \frac{T_1}{RT_1/n + (1 - R)T_1} = \frac{n}{R + n(1 - R)} \tag{1}$$

  となる．

プロセッサ数を増やして並列計算を行っても，並列化できない部分があることにより，プロセッサ数に応じた性能が得られない場合がある．評価実験では実行結果から (1) 式を用いて並列化率を求めた．

3.2　単一ノードでの性能評価

単一ノードでは，ノード内の複数のコア上で動作するプロセスが，共有メモリを用いて互いに計算結果を交換することで並列計算を行う．

3.2.1　単一ノードでのラプラシアン行列の性能評価

まず，行列サイズ 100,000×100,000 のラプラシアン行列を対象とした評価を行った．用いた計算手法は Arnoldi, Krylov-Schur, Lanczos の 3 つである．ここで，最大繰り返し回数を 12,500 回と制限した．この行列サイズでは，この繰り返し回数内にどの計算手法でも固有値が収束せず，最大繰り返し回数まで反復計算が行われる．これにより，3 つの計算手法がそれぞれ同じ回数 (12,500 回) だけ反復計算される．これにより，同じ条件で Arnoldi, Krylov-Schur, Lanczos の結果を比較することができる．

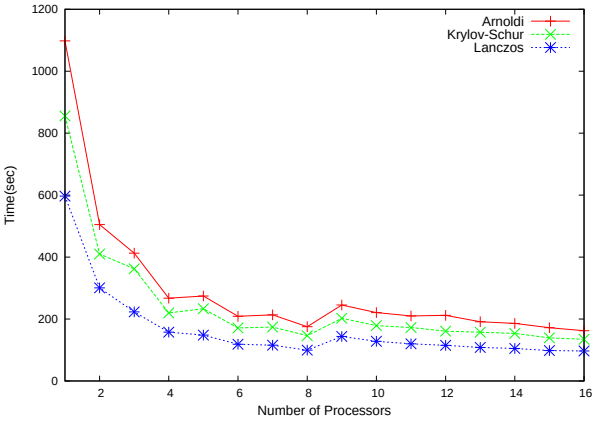

図 5: 単一ノードでのラプラシアン行列における処理時間

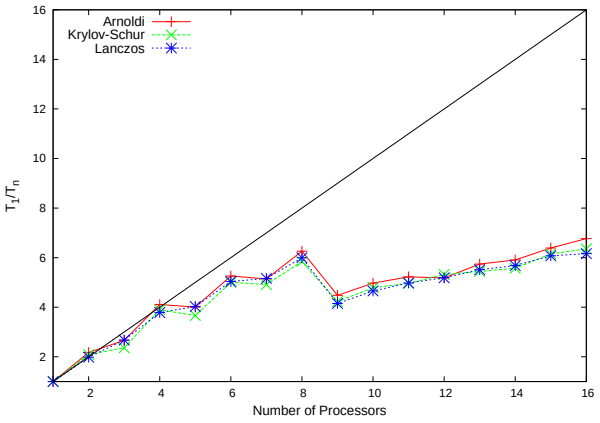

図 6: 単一ノードでのラプラシアン行列における性能向上率 (T_1/T_n)

図 5 は単一ノードでのラプラシアン行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．これより，プロセッサ数が増加するに伴い処理時間も短くなっていることが分かる．対称行列では Lanczos の計算時間が最も短いという結果になった．ただし，これは繰り返し回数を等しくした場合の計算時間であり，実際には各種法で固有値が収束するまでに必要な繰り返し回数は異なっている．行列サイズ 100,000×100,000 のラプラシアン行列において，繰り返し回数を制限せずに固有値が収束するまで反復計算を行ったところ，Arnoldi では 39832 回，Krylov-Schur では 42209 回，Lanczos では 39832 回で収束した．

図 6 は単一ノードでのラプラシアン行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．図 6 中の $y = x$ の直線は理想的な性能を表したものであるが，実際の結果ではアムダールの法則に従い，プロセッサ数の増加に対してすべての計算手法において性能向上率の伸びが鈍化している．また，プロセッサ数が 8 から 9 に変わるところで性能が極端に低下しており，性能向上率においてはプロセッサ数 16 のときの結果がプロセッサ数 8 のときと同等程度の性能しか得られていない．

各計算手法におけるプロセッサ数 8 までの並列化率 R を (1) 式から求めると，Arnoldi では 0.9765, Krylov-Schur は 0.9491, Lanczos では 0.9582 となった．

3.2.2　単一ノードでの遷移確率行列の性能評価

次に，行列サイズ 720,600×720,600 の遷移確率行列の固有値計算にかかる時間を測定した．最大繰り返し回数は 1,000 回に制限してある．遷移確率行列は図 4(b) に示すような非対称行列であるが，ラプラシアン行列よりも対象とする Web ページのリンク構造に近い行列になっている．行列の列方向の値の和が 1 になるように行列を生成するため制約条件があり，行列のサイズは中途半端なものとなる．また，Lanczos は対称行列を対象とした計算手法であるため，非対称行列においては Arnoldi, Krylov-Schur の 2 手法を用いての測定を行った．Krylov-Schur では，行列が対象の場合，部分的に計算アルゴリズムを Lancozs をベースとした計算手法に切り替えているため，非対称の場合では性能特性が異なっている．

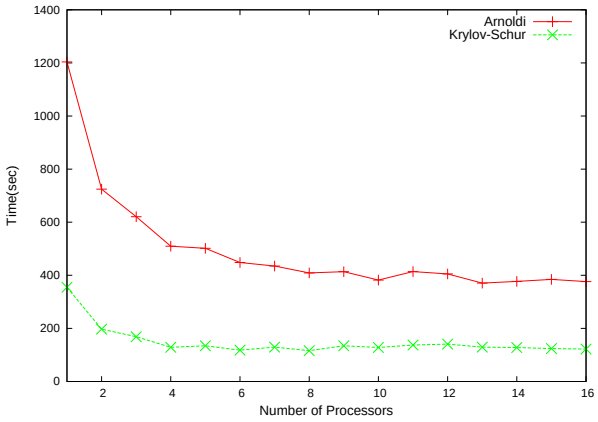

図 7: 単一ノードでの遷移確率行列における処理時間

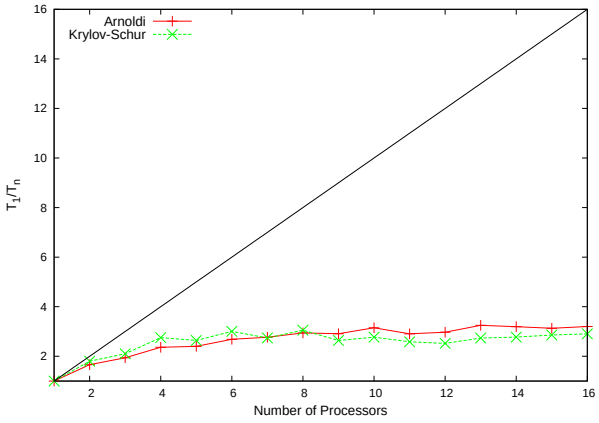

図 8: 単一ノードでの遷移確率行列における性能向上率 (T_1/T_n)

図 7 は単一ノードでの遷移確率行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．Arnoldi と Krylov-Schur の処理時間を比較してみると，Krylov-Schur が Arnoldi に比べかなり処理時間が短いことが分かる．プロセッサ数 8 のときの Arnoldi と Krylov-Schur の処理時間の差は，ラプラシアン行列では 30 秒程度であったが，遷移確率行列ではおよそ 300 秒とかなり大きい．

図 8 は単一ノードでの遷移確率行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．ラプラシアン行列の結果と比べてみるとかなり性能が低いことが分かる．また，各計算手法におけるプロセッサ数 8 までの並列化率 R を (1) 式から求めると，Arnoldi では 0.7534，Krylov-Schur では 0.8015 であった．ラプラシアン行列の並列化率が 90%を超えていたことを考えるとかなり下がっている．また，ラプラシアン行列で見られたプロセッサ数 9 以降の極端な性能低下は見られなかった．

遷移確率行列は疎行列であるため固有値の収束が速く，収束までの繰り返し回数が少ない．実際に収束までにかかった繰り返し回数は，Arnoldi では 1475 回，Krylov-Schur では 1229 回であった．また，プロセッ

サ数 8 のときの処理時間は Krylov-Schur では 116.26 秒であった．ラプラシアン行列と比べ行列サイズがおよそ 7 倍であるにもかかわらず，処理時間は同等程度であった．

3.3　複数ノードでの性能評価

複数ノードを用いた際の性能評価を行った．本実験では Hyper-Threading が有効で 16 プロセッサ利用可能なノードと，無効で 8 プロセッサ利用可能なノードによる 2 ノードでの性能評価を行う．ノード間通信には TCP/IP を用いた．また，計算対象とした行列および条件は単一ノードの場合と同様である．

3.3.1　複数ノードでのラプラシアン行列の性能評価

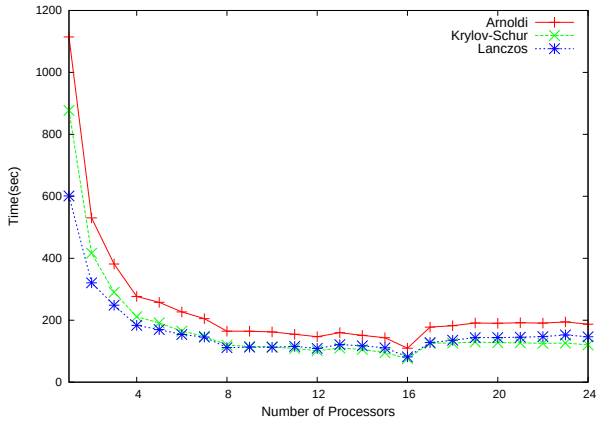

図 9: 複数ノードでのラプラシアン行列における処理時間

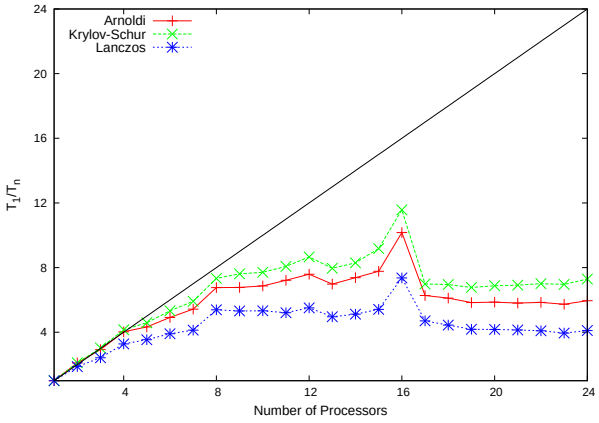

図 10: 複数ノードでのラプラシアン行列における性能向上率 (T_1/T_n)

図 9 は複数ノードでのラプラシアン行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．プロセッサ数 8 までは Lanczos が最も処理時間が短くなっている．しかし，プロセッサ数 8 以降から Krylov-Schur と Lanczos が逆転し，わずかではあるが Krylov-Schur の方が処理時間が短いことが分かる．単一ノードの結果ではこのようなことは見られなかった．これより，Lanczos の性能低下は通信によるオーバヘッドが原因でないかと考えられる．また，最も処理時間が長かったのは，単一ノードの場合と同じく Arnoldi であった．

図 10 は複数ノードでのラプラシアン行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．各計算手法ともプロセッサ数 16 以降性能が極端に低下している．また，単一ノードでは性能向上率にあまり大きな差はなかったが，複数ノードの結果には差があることが分かる．

各計算手法におけるプロセッサ数 16 までの並列化率 R を (1) 式から求めると，Arnoldi では 0.9625，Krylov-Schur では 0.9781，Lanczos では 0.8979 となった．

3.3.2　複数ノードでの遷移確率行列の性能評価

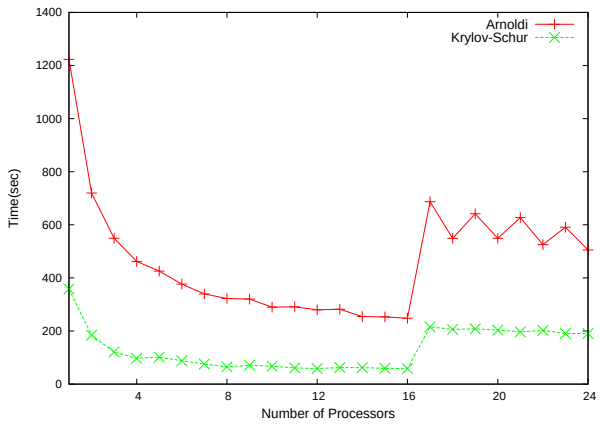

図 11: 複数ノードでの遷移確率行列における処理時間

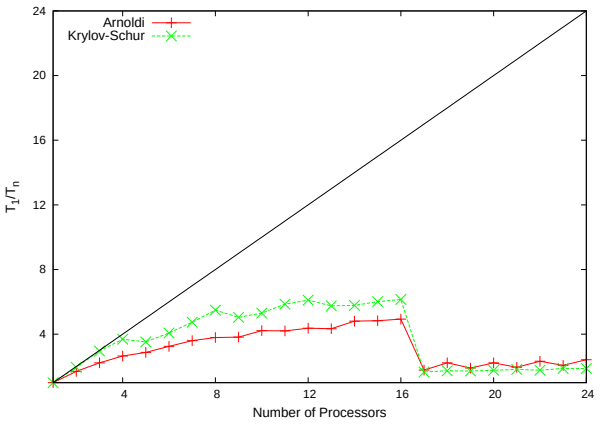

図 12: 複数ノードでの遷移確率行列における性能向上率 (T_1/T_n)

図 11 は複数ノードでの遷移確率行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．プロセッサ数 17 以降の性能が極端に低下していることが分かる．これは単一ノードの結果では見られなかった．プロセッサ数 17 以降の処理時間は Arnoldi ではプロセッサ数 2 のときと同程度，Krylov-Schur ではプロセッサ数 2 のとき以下の性能となっている．また，Arnoldi ではプロセッサ数 17 以降の奇数プロセッサにおいて性能が下がっている．プロセッサ数 17 以降の Krylov-Schur の処理時間はプロセッサ数 16 のときの Arnoldi の処理時間以下である．

図 12 は複数ノードでの遷移確率行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．単一ノードでは Arnoldi と Krylov-Schur の性能向上率にあまり差はなかったが，複数ノードではプロセッサ数 16 までは Krylo-Schur の方が良い結果となった．しかし，プロセッサ数 17 以降は少しであるが Arnoldi の方が良い結果となっている．

各固有値計算手法におけるプロセッサ数 16 までの並列化率 R を (1) 式から求めると，Arnoldi では 0.8368, Krylov-Schur では 0.9195 となった．

3.4　事前評価に伴う考察とその対策

3.4.1　Hyper-Threading による影響

評価環境 agile(表 1) はノードあたり 8 コアのプロセッサが利用可能であるが，Intel Hyper-Threading Technology により，仮想的に 16 コアのプロセッサを利用できる．図 5, 6 より，使用するプロセッサが 8 から 9 に増えるときに性能が低下していることが分かる．これは Hyper-Threading による仮想コアが確実に使われ始めるプロセッサ数 (プロセッサ数 9 以降) と一致している．そこで，プロセッサ数 9 以降の性能低下の原因は Hyper-Threading が原因であると仮定して，BIOS の設定により Hyper-Threading を無効にして再度測

定を行い，Hyper-Threading の有無による性能向上率 (T_1/T_n) を比較した．その結果が図 13 である．ここで，横軸はプロセッサ数，縦軸は性能向上率である．

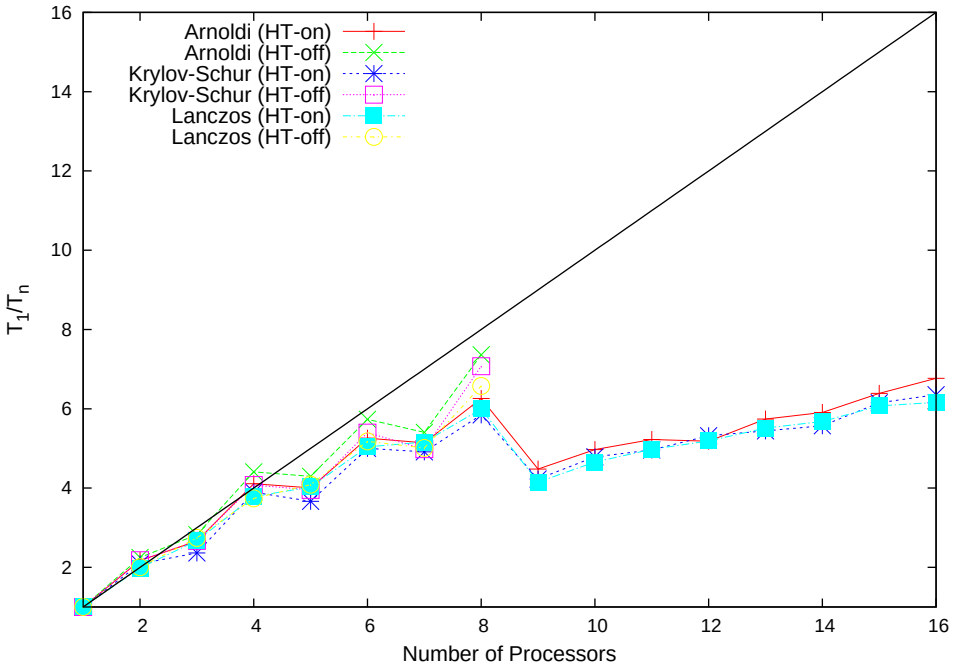

図 13: Hyper-Threading の有無による性能向上率 (T_1/T_n) の比較

どの計算手法においても，Hyper-Threading が無効な場合の結果の方が良いことが分かる．また，有効な場合に見られた極端な性能低下も表れていない．これより，プロセッサ数 9 以降の性能低下の原因は複数 CPU の使用が原因ではなく，Hyper-Threading によるものであるということが分かる．また，プロセッサ数が奇数のとき性能が低下する傾向があるが，これは MPI におけるオールギャザ，オールレデュース等の集合通信では，プロセッサ数が偶数であることを想定しており，奇数の場合余ったノードの通信は効率的に行えなえず，プロセッサ数に応じた性能を得ることができていないためであり，Hyper-Threading が原因ではないと考えられる．

3.4.2　–bind-to-core オプション

評価環境 agile ではプログラムの実行中に使用するプロセッサが頻繁に変更されてしまうことがあり，この頻繁なプロセッサの変更による性能低下があると考えた．(図 14)

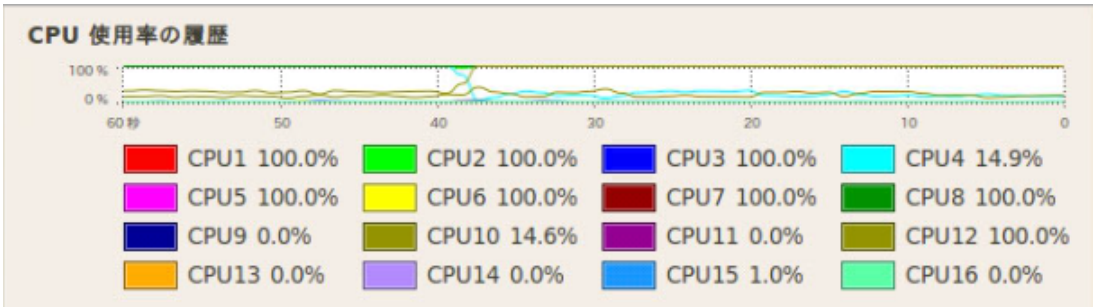

図 14: 実行中の CPU 使用率の履歴

調査の結果，`mpirun` には`--bind-to-core` というオプションがあることが分かった．`--bind-to-core` オプションはその名の通り，各プロセッサに各 MPI プロセスをバインドする．

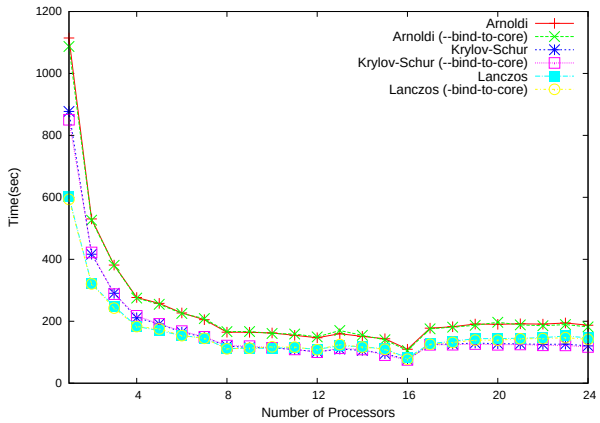

図 15: ラプラシアン行列での–bind-to-core オプションの有無による比較

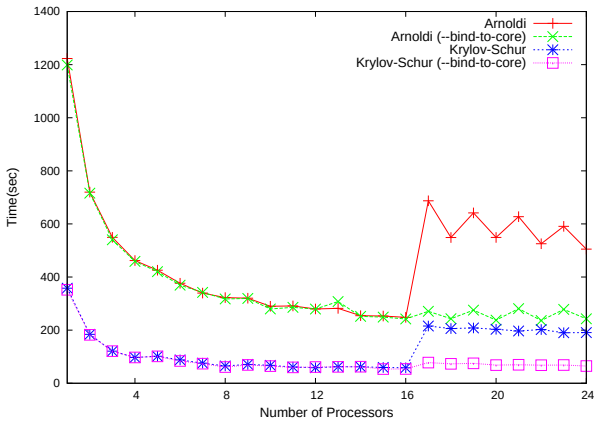

図 16: 遷移確率行列での–bind-to-core オプションの有無による比較

図 15, 16 はそれぞれラプラシアン行列と遷移確率行列の処理時間を`--bind-to-core` オプションの有無で比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．

まず，図 15 はラプラシアン行列の結果であるが，プロセッサ数 1 のときの結果がわずかにであるが`--bind-to-core` オプションを有効にした方が短くなっている．次に，図 16 は遷移確率行列の結果であるが，プロセッサ数 16 まではオプションの有無によってあまり変わらない，しかし，プロセッサ数 17 以降の結果は Arnoldi, Krylov-Schur ともに改善されており，Krylov-Schur では僅かに性能低下が見られるが，Arnoldi ではほぼ Hyper-Threadig の影響がない．

Hyper-Threading による性能低下の影響を受けるプロセッサ数 17 以降の結果が特に変化していることより，

実行中の頻繁なプロセッサの変更は Hyper-Thraeding が原因であったと考えられる．そして，`--bind-to-core` オプションを有効にすることで，遷移確率行列においてはその影響を改善することができた．

以降，`--bind-to-core` オプションを用いることとする．

3.4.3　通信によるオーバヘッドの影響

図 17 は，ラプラシアン行列での Krylov-Schur における単一ノードと複数ノードの処理時間を比較したものである．単一ノードのプロセッサ数 9 以降，複数ノードのプロセッサ数 17 以降の結果が極端に悪くなっているのは Hyper-Threading によるものであることが分かっている．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．単一ノードと複数ノードの結果を比較したとき，同じプロセッサ数であれば TCP/IP 通信によるオーバヘッドがある複数ノードの方が処理時間が長くなると想定していた．しかし，図 17 よりわずかにであるが，複数ノードの処理時間に比べ単一ノードの方が長くなっていることが分かる．

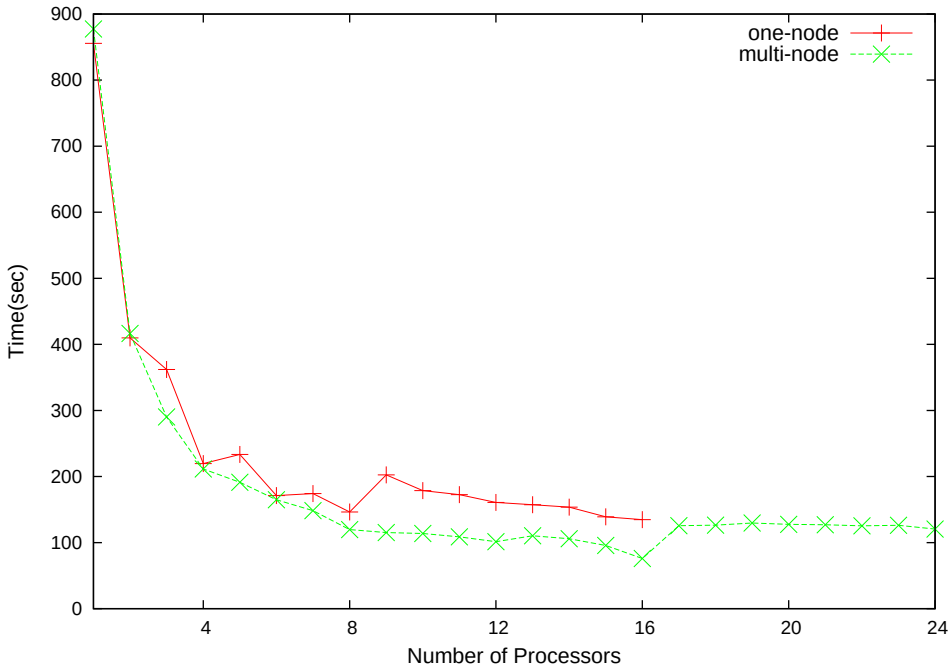

図 17: ラプラシアン行列での Krylov-Schur における単一ノードと複数ノードの処理時間の比較

通信によるオーバヘッドの影響を調べるために，ラプラシアン行列での Krylov-Schur を用いた調査を行った．遷移確率行列は疎行列であるため固有値の収束が速く，繰り返し回数が少なくなるため，繰り返し回数が多くなるラプラシアン行列を用いた．繰り返し回数を 6250, 9375, 12500, 18750, 25000 回と変化させたときの単一ノードと複数ノードの処理時間を比較し，通信によるオーバヘッドの影響を調査する．

図 18 は 1 ノードで 2 プロセッサ使用したときの処理時間と 2 ノードで 1 プロセッサずつ使用したときの処理時間を，図 19 は 1 ノードで 8 プロセッサ使用したときの処理時間と 2 ノードで 4 プロセッサずつ使用したときの処理時間を比較したものである．ここで，横軸は繰り返し回数，縦軸は処理時間 (秒) である．

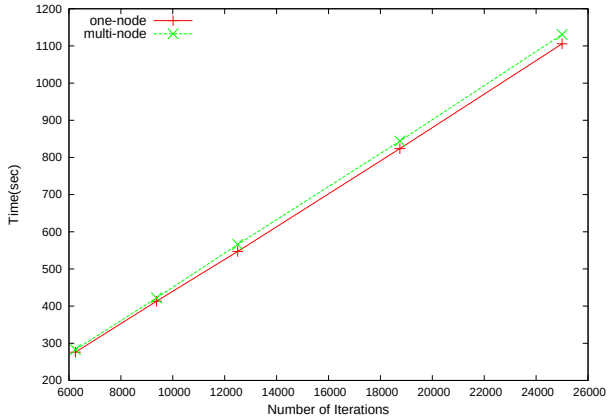

図 18: 2 プロセッサ使用したときの通信によるオーバヘッドの影響

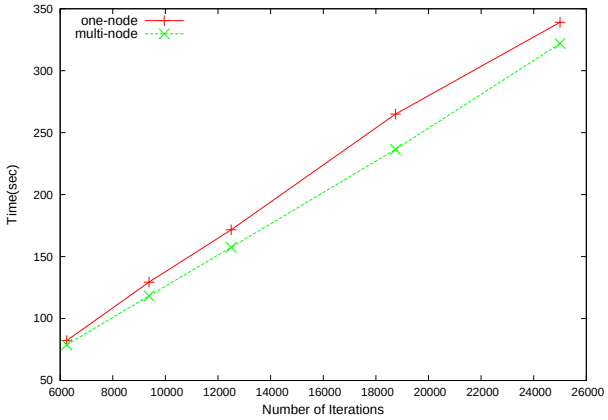

図 19: 8 プロセッサ使用したときの通信によるオーバヘッドの影響

まず，プロセッサを 2 つ使用したときであるが，図 18 より，単一ノードの処理時間に比べ複数ノードの処理時間の方が長くなっていることがわかる．これより，確かに複数ノードを用いた際に通信によるオーバヘッドが生じていることが分かる．次に，プロセッサを 8 つ使用したときであるが，図 19 より，複数ノードの処理時間に比べ単一ノードの処理時間の方が長くなっていることがわかる．単一ノードでの並列計算を行うとき，プロセス間の通信は共有メモリを介して行われる．

これらの結果より，複数ノードを用いた際の通信によるオーバヘッドによる影響は確かにあることが分かった．しかし，使用するプロセッサを増やしていくと，単一ノードの処理時間の方が複数ノードに比べ長くなる．これは，通信によるオーバヘッドよりも複数プロセスが共有メモリにアクセスする際の入出力 (I/O) 待ちの影響が大きくなっているのではないかと考えられる．

4　ランキング計算手法の評価 (CentOS, HP) と異種環境間での性能比較

より大規模な計算について評価するため，評価環境 social(表 2) を新たに用意した．social において agile と同様の性能評価を行い，それらを比較することでこれまで発見されている問題点を別の視点から調査する．agile での Hyper-Threading による性能低下の結果を受け，social の Hyper-Threading は無効にしている．

4.1　単一ノードでの性能評価と比較

4.1.1　単一ノードでのラプラシアン行列の性能評価と比較

まず，ラプラシアン行列を用いた性能評価を行い，agile の結果との比較を行った．この際の条件は agile の場合と同様に，行列サイズ 100,000×100,000, 最大繰り返し回数 12,500 回である．

図 20 は単一ノードでのラプラシアン行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．すべての計算手法において，agile と social を比較すると，social の

表 2: 評価環境 social

計算機	HP ProLiant SL390s G7 2U ×4
CPU	Intel(R) Xeon(R) X5650 (2.67GHz, 12MB キャッシュ, 6.40GT/s QPI) ×2 (コア数 6)
メモリ	24GB (4GB×6/2R/1333MHz/DDR3 RDIMM)
ディスク	500GB 7200PRM SATA
ネットワーク	NC362i dual-port Gigabit Ethernet
スイッチ	Cisco SG300-20-JP
OS	CentOS release 5.5 (Final)
ライブラリ	PETSc 3.1-p8, SLEPc 3.1-p6, FBLASLAPACK 3.1.1 OpenMPI 1.4.2

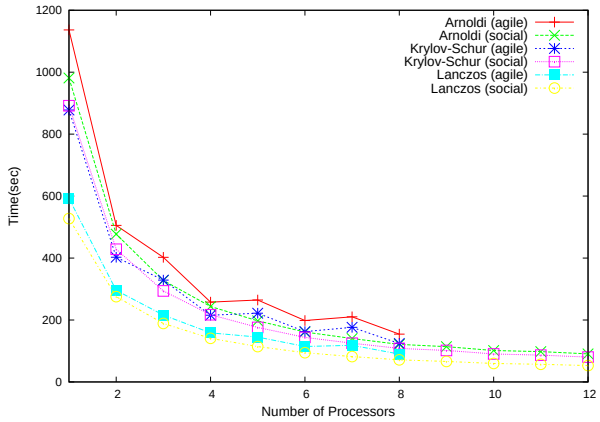

図 20: 単一ノードでのラプラシアン行列における処理時間の比較

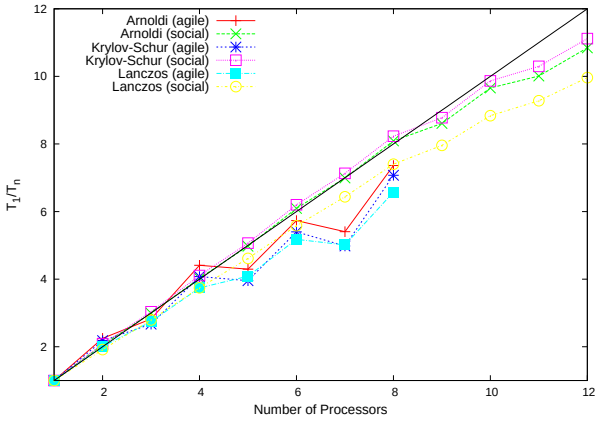

図 21: 単一ノードでのラプラシアン行列における性能向上率 (T_1/T_n) の比較

方が良い結果となった．agile の結果と同様に，Lanczos が最も処理時間が短い．また，agile ほど奇数プロセッサでの性能低下がないことが分かる．

図 21 は単一ノードでのラプラシアン行列における性能向上率 (T_1/T_n) を比較したものである．図中の $y = x$ の直線は理想的な性能を表したものであるが，social での結果は，プロセッサ数 9 以降から少し低下するが，プロセッサ数 8 までの Arnoldi, Krylov-Schur の結果は，ほぼこの直線に等しいかそれ以上であり，かなり良いことが分かる．また，Lanczos も Arnoldi と Krylov-Schur ほどでないが良い結果を示している．agile の結果と比較するとその性能向上率の高さが良くわかる．

各計算手法の並列化率 R を (1) 式から求めると，Arnoldi は 1.0002，Krylov-Schur は 1.0047, Lanczos は 0.9787 となり，すべての計算手法においてかなり良い結果となった．

4.1.2　単一ノードでの遷移確率行列の性能評価と比較

次に，遷移確率行列を用いた性能評価を行い，agile の結果との比較を行った．この際の条件は agile の場合と同様に，行列サイズ 720,600×720,600, 最大繰り返し回数 1,000 回である．

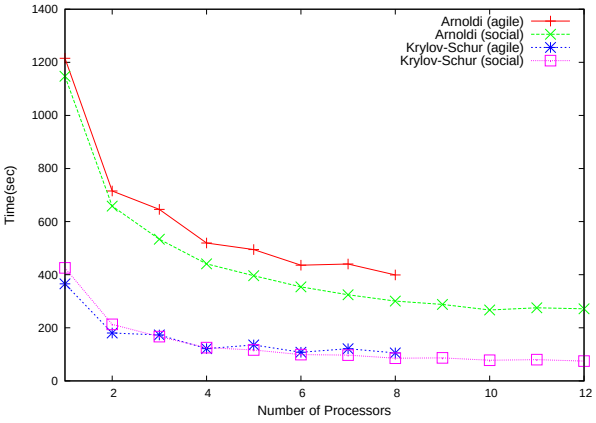

図 22: 単一ノードでの遷移確率行列における処理時間の比較

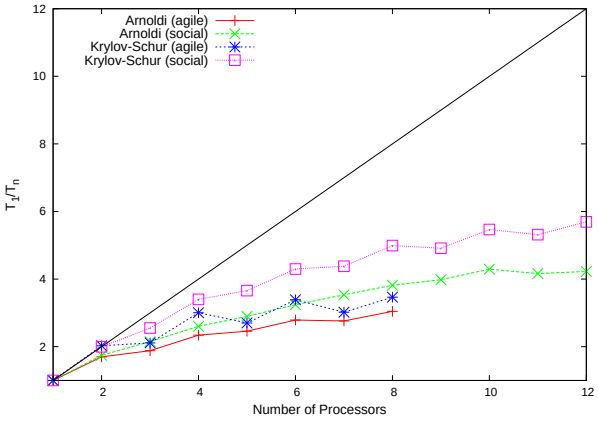

図 23: 単一ノードでの遷移確率行列における性能向上率 (T_1/T_n) の比較

図 22 は単一ノードでの遷移確率行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．Arnoldi においては social の方が良い結果を示した．また，Krylov-Schur も僅かであるが social の方が良い結果となった．また，Arnoldi と Krylov-Schur の処理時間の差は，agile の場合よりは小さくなったものの，プロセッサ数 8 の場合において約 200 秒程度と大きいことが分かる．

図 23 は単一ノードでの遷移確率行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．agile と social の性能向上率を比較すると，Arnoldi, Krylov-Schur 共に social の方が良い結果となった．また，ラプラシアン行列ではあまり表れなかった奇数プロセッサでの性能低下が発生していることが分かる．

単一ノードでの性能評価の傾向としてはラプラシアン行列，遷移確率行列ともに social の方が性能が良いという結果になった．特に性能向上率において，social が agile に比べ良い結果となった．

4.2　複数ノードでの性能評価と比較

複数ノードでのラプラシアン行列，遷移確率行列の性能評価を行う．条件等は単一ノードと同様である．なお，比較の対象となる agile は，1 ノードが Hyper-Threading が有効，もう 1 ノードが無効となっている．

4.2.1　複数ノードでのラプラシアン行列の性能評価と比較

図 24 は複数ノードでのラプラシアン行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．social の結果は，プロセッサ数 1 のとき，Lanczos が最も短かった．

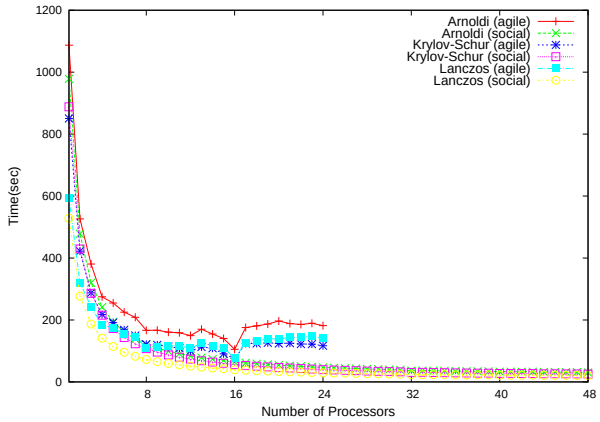

図 24: 複数ノードでのラプラシアン行列における処理時間の比較

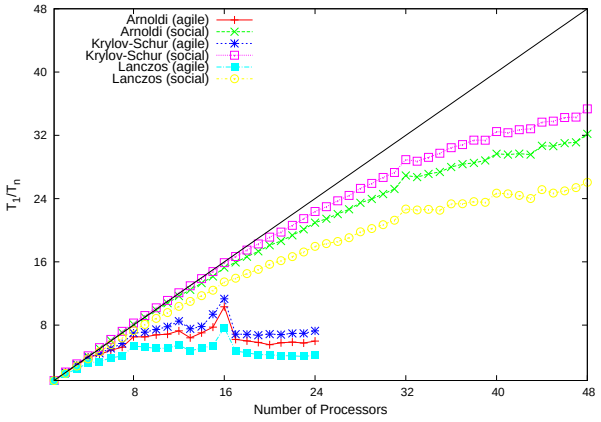

図 25: 複数ノードでのラプラシアンにおける性能向上率 (T_1/T_n) の比較

しかし，プロセッサ数が増加するに伴い，Arnoldi, Krylov-Schur, Lanczos の差はほぼなくなり，プロセッサ 24 以降ほどから処理時間もあまり短縮できてないことが分かる．agile の結果と比較すると，Krylov-Schur でのプロセッサ数 1 から 4 までの結果を除き，social の方が処理時間が短くなっている．

図 25 は複数ノードでのラプラシアン行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．Arnoldi, Krylov-Schur の性能向上率は，プロセッサ数 10 まで理想的な直線とほぼ等しい．また，プロセッサ数 11 以降も極端な性能低下は見られず，32, 40 といった 8 の倍数のとき性能が良くなっていることが分かる．agile の結果と比較すると，すべての計算手法において，social が良いという結果になった．

4.2.2 複数ノードでの遷移確率行列の性能評価の比較

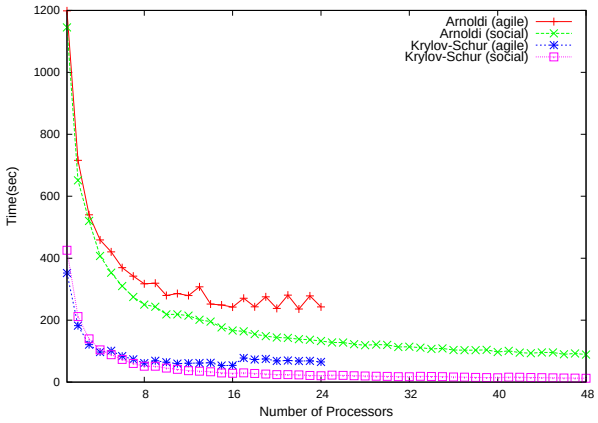

図 26: 複数ノードでの遷移確率行列における処理時間の比較

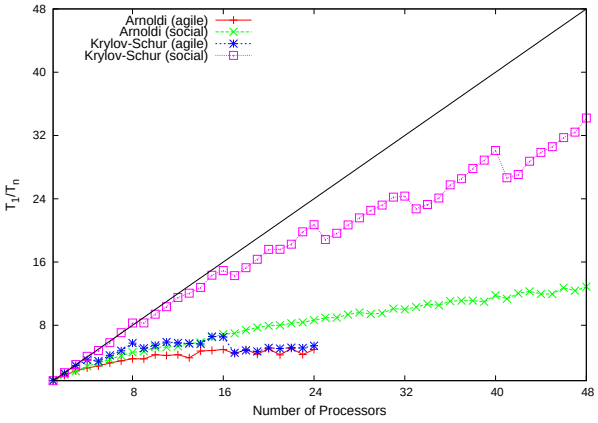

図 27: 複数ノードでの遷移確率行列における性能向上率 (T_1/T_n) の比較

図 26 は複数ノードでの遷移確率行列における処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．agile の結果と比較すると，social がほとんどの場合において良い結果となっている．しかし，Krylov-Schur ではプロセッサ数 16 以降ごろからプロセッサ数が増加しているにもかかわらず，処理時間が平行線となり，あまり短縮できてないことが分かる．

図 27 は複数ノードでの遷移確率行列における性能向上率 (T_1/T_n) を比較したものである．ここで，横軸はプロセッサ数，縦軸は性能向上率である．agile の結果と比較すると，social での Krylov-Schur の結果がかなり良い性能を示していることが分かる．また，$8n+1$ プロセッサでの性能が下がっているが，これは MPI におけるオールギャザ，オールレデュース等の集合通信が偶数であることを想定しているためである．

4.3 social における通信によるオーバヘッドの影響

agile では，通信によるオーバヘッドよりも複数プロセスが共有メモリにアクセスする際の入出力待ちの影響が大きくなっていた (3.4.3 節)．social でも同様に，通信によるオーバヘッドの影響を調査した．

図 28 は，social においてラプラシアン行列での Krylov-Schur における単一ノードと複数ノードの処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．これより，プロセッサ数の増加に伴い，単一ノードの処理時間の方が長くなっていることが分かる．しかし，その差は僅かであり，プロセッサ数 8 まではほぼないことが分かる．単一ノードでの並列計算を行う際，プロセス間の情報のやり取りは共有メモリを介して行われる．そこで，Ubuntu と CentOS のカーネルパラメータを比較したところ，共有メモリの最大サイズに関わると思われる `kernel.shmmax` というカーネルパラメータの値が agile では 33554432 であったのに対し，social では 68719476736 とかなり差があることが分かった．そこで，agile の `kernel.shmmax` を 68719476736 に変更し，agile における通信によるオーバヘッドの影響を再調査した．

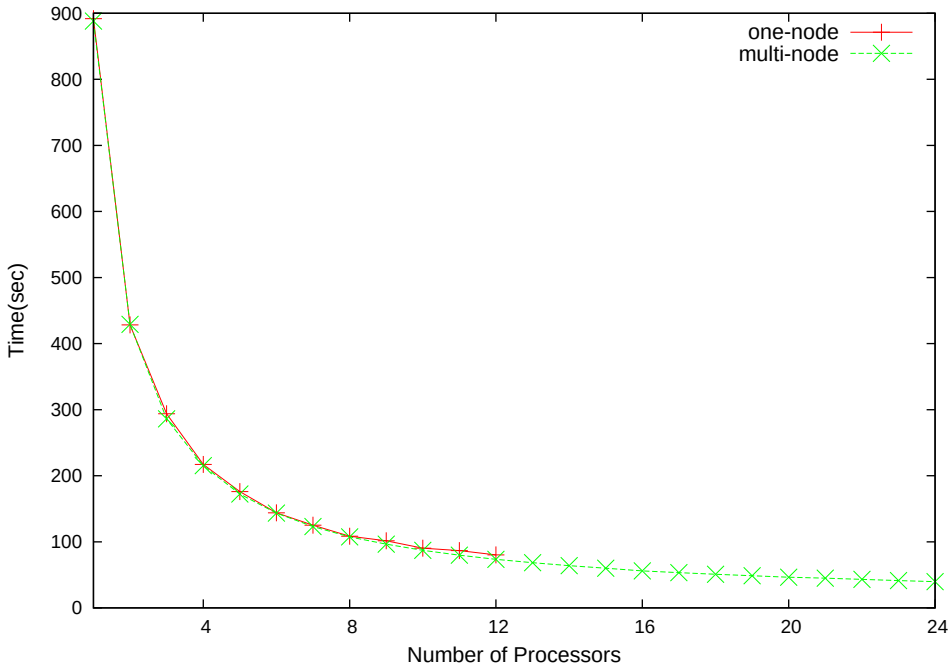

図 28: social における単一ノードと複数ノードの処理時間の比較

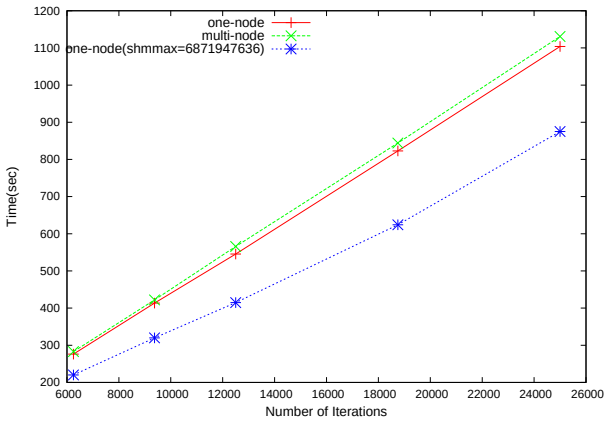

図 29: 2 プロセッサ使用したときの通信によるオーバヘッドの影響

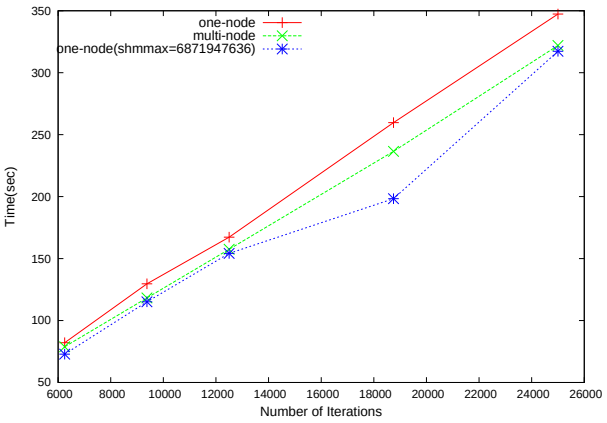

図 30: 8 プロセッサ使用したときの通信によるオーバヘッドの影響

図 29 は 2 プロセッサ使用したときの処理時間を，図 30 は 8 プロセッサ使用したときの処理時間をそれぞれ単一ノード，複数ノード，単一ノード (共有メモリの最大サイズ変更) で比較したものである．ここで，横軸は繰り返し回数，横軸は処理時間 (秒) である．まず，図 29 であるが，処理時間が最も長かったのは複数ノードによる結果である．これは通信によるオーバヘッドの影響があるためである．また，単一ノードにおいて共有メモリの最大サイズを変更した場合の結果はデフォルトに比べて処理時間が短いことが分かる．

次に，図 30 であるが，単一ノードと複数ノードの処理時間を比較すると，複数ノードの方が短い．これは，通信によるオーバヘッドよりも複数プロセスが共有メモリにアクセスする際の入出力待ちの影響が大きくなっているためであると仮定している (3.4.3 節)．また，単一ノードにおいて共有メモリの最大サイズを変更した場合の処理時間は複数ノードの結果よりも短いことが分かる．

これらより，プロセッサ数を増加させたときに，複数ノードでは通信によるオーバヘッドの影響があるにもかかわらず，単一ノードでの処理時間の方が長かったのは，複数プロセスが共有メモリにアクセスする際の入出力待ちが原因であることが分かった．そして，共有メモリの最大サイズを変更することでそれを解消することができる．

4.4 考察

agile と social の結果を比べると，ほとんどの場合において social が良い結果となった．表 3 に示した通り，agile と social にはハードウェア環境の違いがある．しかし，ハードウェア環境による性能差を考慮しても，想定していたより agile と social の性能に差が表れていた．ソーシャルサーチシステムは企業等のプライベートな情報システムに対しても適用することを考えており，どのような環境でもより良いサービスを提供するために，この性能差の原因を調査し，解消する必要がある．以下ではこの性能差の原因を調査し，可能であればその解消を行う．

表 3: 評価環境の比較

	agile	**social**
計算機	DELL PowerEdge R410 ×2	HP ProLiant SL390s G7 2U ×4
CPU	Intel Xeon L5520 (2.26GHz, 8MB キャッシュ, 5.86 GT/s QPI) ×2 (コア数 4)	Intel Xeon X5650 (2.67GHz, 12MB キャッシュ, 6.40GT/s QPI) ×2 (コア数 6)
メモリ	32GB (4GB×8/2R/1066MHz/DDR3 RDIMM)	24GB (4GB×6/2R1333MHz/DDR3 RDIMM)
ディスク	RAID6 (PERC6i), 600GB, 15,000RPM SAS	500GB 7200PRM SATA
ネットワーク	Broadcom NetXtreme II BCM5716 1000Base-T PCI Express	NC362i dual-port Gigabit Ethernet
スイッチ	Cisco Catalyst 2960G-8TC-L	Cisco SG300-20-JP
OS	Ubuntu Linux 10.04	CentOS release 5.5 (final)
ライブラリ	PETSc 3.0.0, SLEPc 3.0.0, LAPACK 3.2.1, BLAS 1.2, OpenMPI 1.4.1	PETSc 3.1-p8, SLPEc 3.1-p8 FBLASLAPACK 3.1.1, OpenMPI 1.4.2

5 異種環境間での性能差の原因調査

以下では agile と social の性能差の原因調査を行う．

5.1 使用するライブラリのバージョンについて

表 3 は agile と social の評価環境をを比較したものであるが，agile では PETSc3.0.0 と SLEPc3.0.0 を，social では PETSc3.1 と SLEPc3.1 を利用しており，agile と social で利用する PTESc と SLEPc のライブラリのバージョンが異なることが分かる．

5.1.1 関数の呼び出し回数の比較

PETSc では`-log_summary` オプションを用いると関数の呼び出し回数を得ることができる．これを用いて agile と social の結果を比較したところ，呼び出し回数が異なる関数がいくつかあった．例として，Krylov-Schur を用いた際のラプラシアン行列での結果を比較したものを表 4 に，遷移確率行列での結果を比較したものを表 5 に示す．以下では PETSc3.0, SLEPc3.0 を用いるものを Ver3.0 と，PETSc3.1, SLEPc3.1 を用いるものを Ver3.1 と称する．

表 4: ラプラシアン行列での呼び出し回数が異なる関数一覧

	Ver3.0	Ver3.1
EPSDense	-	12500
IPOrthogonalize	100008	100009
VecAXPY	212514	12499
VecMAXPY	200015	-
VecMAXPBY	-	200015

表 5: 遷移確率行列での呼び出し回数が異なる関数一覧

	Ver3.0	Ver3.1
STApply	7679	7562
EPSDense	3999	2999
IPOrthogonalize	7679	7564
IPInnerProduct	30720	30256
VecScale	7680	7564
VecMult	7679	7562
VecAXPY	7680	-
VecMAXPY	7680	-
VecMAXPBY	-	7564

表 4, 5 よりライブラリのバージョンによって関数の呼び出し回数が異なり，呼び出されていない関数もあることが分かる．これより，ライブラリのバージョンの違いが agile と social の性能差の原因のひとつであるのではないかと考えた．

5.1.2 使用するライブラリのバージョンの変更

agile と social の性能差の原因はライブラリのバージョンの違いによるものであると仮定した．そこで，それを確認するために agile に Ver3.1 をインストールし，Ver3.0 と Ver3.1 の処理時間を比較した．この際の条件はラプラシアン行列は行列サイズ 100,000×100,000, 最大繰り返し回数 12,500 回とし，遷移確率行列は行列サイズ 720,600×720,600，最大繰り返し回数 1,000 回とした．

図 31 はラプラシアン行列において，図 32 は遷移確率行列においてバージョンの違いによる処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．

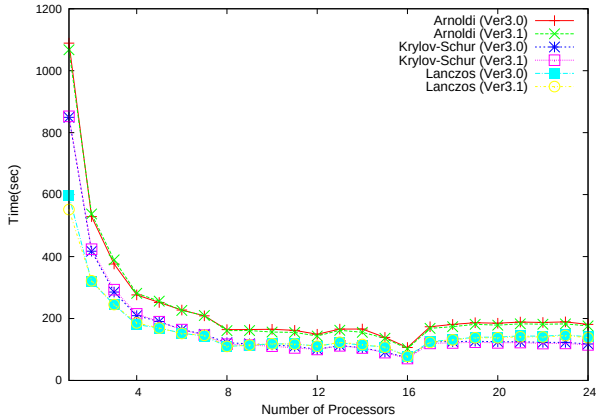

図 31: ラプラシアン行列でのライブラリのバージョンの違いによる処理時間の比較

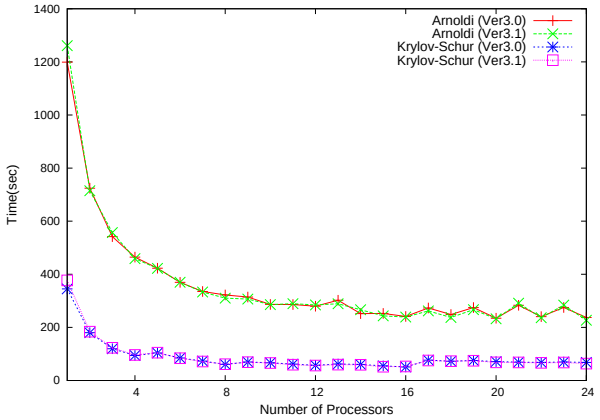

図 32: 遷移確率行列でのライブラリのバージョンの違いによる処理時間の比較

まず，ラプラシアン行列についてであるが，プロセッサ数 1 のとき，Arnoldi はあまり変わらないが，Krylov-Schur と Lanczos は Ver3.1 の方が僅かに処理時間が短いことが分かる．しかし，プロセッサ数 2 以降では両者にあまり大きな差は見られなかった．これより，ラプラシアン行列においてライブラリのバージョンの違いによる性能差はプロセッサ数 1 の場合を除いてあまりないことが分かる．次に遷移確率行列についてであるが，プロセッサ数 1 のとき，Ver3.0 の方が僅かに処理時間が短いことが分かる．しかし，プロセッサ数 2 以降では両者にあまり大きな差は認められなかった．これより，遷移確率行列においてもライブラリのバージョンの違いによる性能差はプロセッサ数 1 の場合を除いてあまりないことが分かる．

よって，ライブラリのバージョンの違いにより僅かに差はあるものの，agile と social の性能差の決定的な原因でないことが分かった．

5.2　仮想マシンを用いた調査

評価環境 agile と social は表 3 に示した通りハードウェア性能が異なる．そこで，仮想マシンを用いてハードウェア環境と利用するライブラリ等のソフトウェア環境を統一した上で Ubuntu と CentOS の比較を行った．仮想マシンを用いた結果を Ubuntu と CentOS で比較し，そこに性能差がなければ agile と social の性能差の原因はハードウェア性能の違いによるものとなり，性能差があれば Ubuntu と CentOS という OS の違いによるものとなる．

仮想マシンには VirtualBox 4.1.6 を用いた．ホスト OS の環境は表 6 に示すものとし，ゲスト OS には Ubuntu 10.04, CentOS5.5 を用いて比較を行った．仮想マシンの設定はプロセッサ数 6，メモリ 2GB とした．また，使用したライブラリは PETSc3.1-p8, SLEPc3.1-p6, OpenMPI 1.4.1 である．

図 33 はラプラシアン行列を，図 34 は遷移確率行列を仮想マシンを用いて評価環境を統一した上で Ubuntu と CentOS の処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．

図 33, 34 より，仮想マシンを用いて環境を統一したにも関わらず，Ubuntu と CentOS の処理時間には差が生じていることが分かる．特に Krylov-Schur はラプラシアン行列，遷移確率行列ともに差が大きいことが分かる．

表 6: 評価環境 iMac

計算機	Apple iMac (Mid 2010)
CPU	Intel Core i7-870 (8MB キャッシュ，2.5GT/s QPI) (コア数 4)
メモリ	8GB 1333MHz DDR3
OS	Mac OS X 10.6.8 (Snow Leopard)

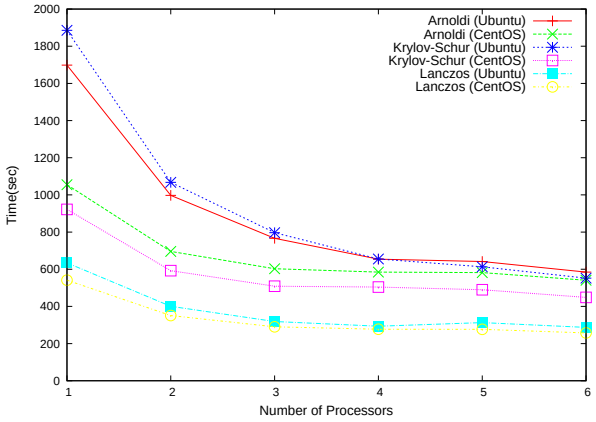

図 33: 仮想マシンを用いたラプラシアン行列の処理時間の比較

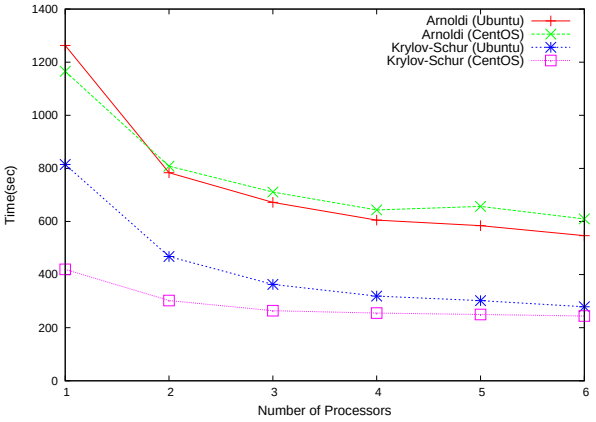

図 34: 仮想マシンを用いた遷移確率行列の処理時間の比較

これより，仮想マシンを用いてハードウェア環境を統一し，同じライブラリを使用したのも関わらず，Ubuntu と CentOS の結果に差が生じたことより，agile と social の性能差の原因はハードウェア性能でなく，Ubuntu と CentOS という OS の違いによるものであると考えられる．

以降，特に Ubuntu と CentOS で性能差が大きかった遷移確率行列での Krylov-Schur を中心に調査を行う．これは，遷移確率行列は提案システムで必要となる固有値計算の対象となる行列に似た性質を持つことと，Arnoldi と Krylov-Schur を比較した際に Krylov-Schur の方が処理時間が短いことから，最終的に提案システムに SLEPc を用いる場合，遷移確率行列での Krylov-Schur の結果が重要となると考えられるからである．

5.3 Ubuntu と CentOS の性能差の原因調査

評価環境 agile と social の性能差の原因は Ubuntu と CentOS という OS の違いにあるということが分かった．Ubuntu と CentOS の性能差の原因を調査するために以下の調査を行った．

5.3.1 sysctl を用いた調査

まず，`sysctl` を用いた調査を行った．`sysctl` はカーネルパラメータを実行時に修正するのに用いられる．また，`-a` オプションをつけることで設定されているカーネルパラメータの一覧とその値を得ることができる．Ubuntu と CentOS で共に設定されているカーネルパラメータを比較することで性能差の原因を調査した．`sysctl` によって得られたカーネルパラメータを一部抜粋し Ubuntu と CentOS で比較したものが表 7 である．

表 7: Ubuntu と CentOS のカーネルパラメータの比較

	Ubuntu	CentOS
kernel.threads-max	32077	65534
kernel.shmmax	33554432	4294967295
kernel.shmmni	2097152	268435456
kernel.msgmax	8192	65536

これらは特にカーネルパラメータの値の差が大きいものである．特に `kernel.shmax` は共有メモリの最大値であるが，CentOS の値がかなり大きいことが分かる．単一ノードでの計算では共有メモリを介した並列計算を行っているので，このようなメモリに関するカーネルパラメータは Ubuntu と CentOS の性能差に関係していると考えた．そこで，`sysctl` を用いて Ubuntu のカーネルパラメータを CentOS の値に変更し，再度評価を行い CentOS の結果と比較してみたが，変更する前と比べあまり大きな差は認められなかった．

`kernel.shmmax` は共有メモリの最大サイズであるが，カーネルパラメータを 4294967295bytes(4GB) に変更したとしても，仮想マシンの設定でメモリを 2GB しか取っていないので，共有メモリを 4GB 使用することはできない．そのためカーネルパラメータを変更してもあまり影響はなかったのだと考えられる．

5.3.2 strace と htop を用いた調査

次に，`strace` と `htop` を用いて，システムコールとプロセスの関係の面からの調査を行った．

`strace` はプログラムが使用するシステムコールおよび受け取るシグナルを監視するものである．`strace` を用いることで実行中にプログラムが使用するシステムコールの呼び出し回数やそれにかかる時間を得ることができる．`strace` を用いて実行中に呼び出されるシステムコールから Ubuntu と CentOS の性能差の原因を調査した．

`strace` による結果を見てみると呼び出されているシステムコールの中では `poll` が最も時間を取っていた．ここで `poll` はファイルディスクリプタにおけるイベントを待つシステムコールである．

htop はインタラクティブなプロセスビューワであるが，使用中に t を押すことでプロセスの親子関係を図 35, 36 のようにツリー状に表示することができる．

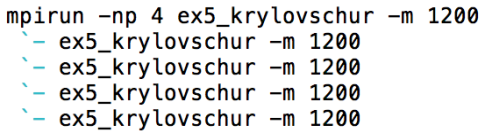

図 35: Ubuntu でのプロセスの関係

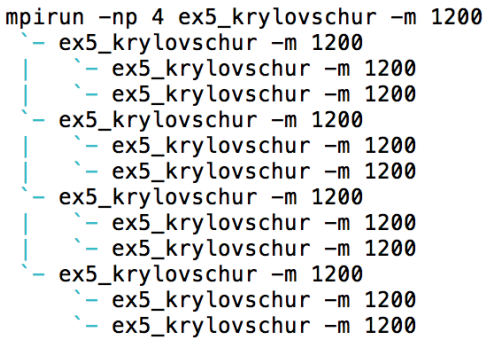

図 36: CentOS でのプロセスの関係

図 35, 36 はプロセッサ数 4 のときの htop の結果を抜粋したものであるが，これらを見てみるとともに mpirun という親プロセスがあり，その親プロセスが子プロセスを作成することにより並列計算を行っていることが分かる．また，図 36 より CentOS では子プロセスがさらに子プロセスを 2 個ずつ持っていることが分かる．これは Ubuntu では確認できなかった．

strace は-f オプションをつけることで子プロセスが呼び出すシステムコールも得ることができる．そこで strace -f を用いて子プロセスが呼び出すシステムコールも含めて再度調べてみると，CentOS では poll に加え select も呼び出されていることが分かった．select も poll と同様にファイルディスクリプタにおけるイベントを待つシステムコールである．strace -f により子プロセスのシステムコールも取得したところ select が増えたことから，CentOS での子プロセスの子プロセスでは select が使われているのではないかと考えた．そこで，CentOS の子プロセスの子プロセスを GDB を用いて調査したところ，2 つのうち 1 つは poll を呼び出した状態で，もう 1 つは select を呼び出した状態で停止していた．また，プロセスの状態も監視してみると sleeping となっており，CPU 使用率も 0%であった．

システムコールの違いと図 35, 36 のようなプロセスの関係の違いが Ubuntu と CentOS の性能差に関わっている考えていたが，strace, htop, GDB を用いた調査の結果から現段階ではこの違いによる性能差はないと考えている．

5.3.3 -log_summary オプションを用いた調査

PETSc では実行時に-log_summary オプションを適用することで，事前設定した関数の実行時間や呼び出し回数を得ることができる．これを用いて実行時に呼び出される関数の実行時間を比較し，プログラム中のどの関数が Ubuntu と CentOS の性能差に関わっているのかを調査した．

図 37 はプロセッサ数 4 のときの遷移確率行列で Krylov-Schur を用いた際に，-log_summary オプションによって得られたログを用い，呼び出された各関数の処理時間を Ubuntu と CentOS で比較したものである．ここで，横軸は呼び出された関数名，縦軸はその処理時間 (秒) である．なお，STSetUp や EPSDense

などグラフが表示されていない関数がいくつかあるが，これは処理時間が極めて短いためである．また，複数の関数で同様な差が見られるのは関数間の包含関係によるものである．

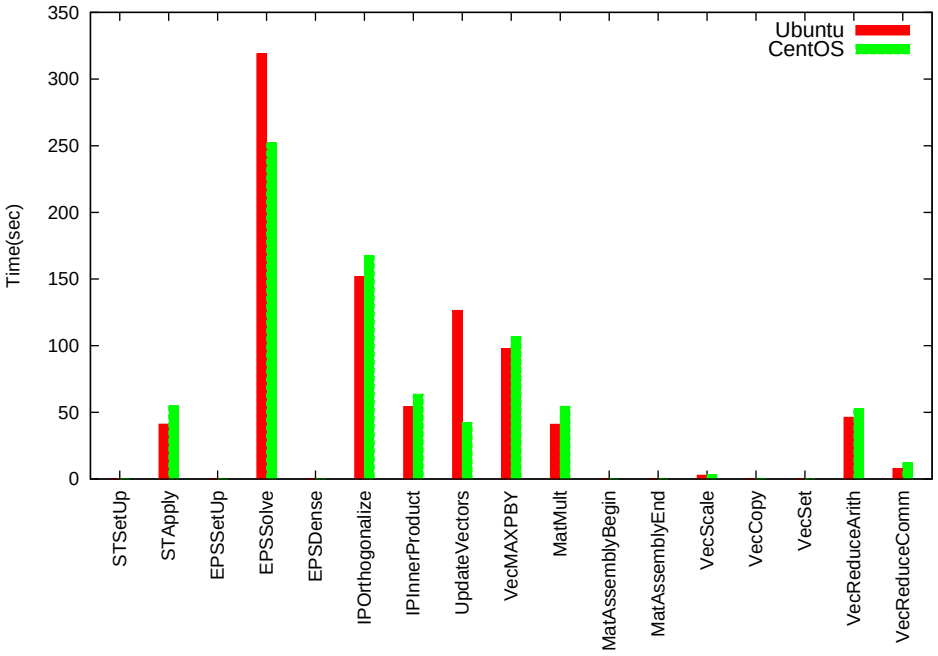

図 37: 遷移確率行列での Krylov-Schur の各関数における処理時間の比較

図 37 より IPOrthogonalaize や IPInnerProduct などといった関数は Ubuntu の方が処理時間が短いことが分かる．しかし，UpdateVectors を見てみると，Ubuntu に比べ CentOS の処理時間がかなり短く半分以下である．`EPSSolve` は全体として固有値計算にかかった時間であるが，これまでの調査から分かっている通り，遷移確率行列での Krylov-Schur の処理時間は CentOS の方が短い．Ubuntu の方が処理時間が短い関数がいくつかあるにもかかわらず，全体として CentOS の方が処理時間が短いことから，UpdateVectors が Ubuntu と CentOS の性能差の原因になっていることが分かる．

5.3.4 gprof を用いた調査

`-log_summary` オプションを用いた調査により，プログラム中のどの部分が性能差の原因となっているのか見当をつけることができた．この結果をふまえ，`gprof` を用いた関数に関するさらに詳細な調査を行う．

`gprof` はプロファイラの一種である．`gprof` を用いることでプログラム実行時に関数の呼び出し回数やそれにかかる時間を得ることができる．`gprof` を用いた調査により，遷移確率行列での Krylov-Schur では `dgemm` と `dgemv` という関数が処理時間の多くを占めていた．

図 38 は全体の処理時間 (折れ線グラフ) と `dgemm+dgemv` の処理時間 (棒グラフ) を Ubuntu と CentOS で比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．これより，`dgemm` と `dgemv` の処理時間を足したものが全体の処理時間の半分以上を占めていることが分かる．また，Ubuntu と

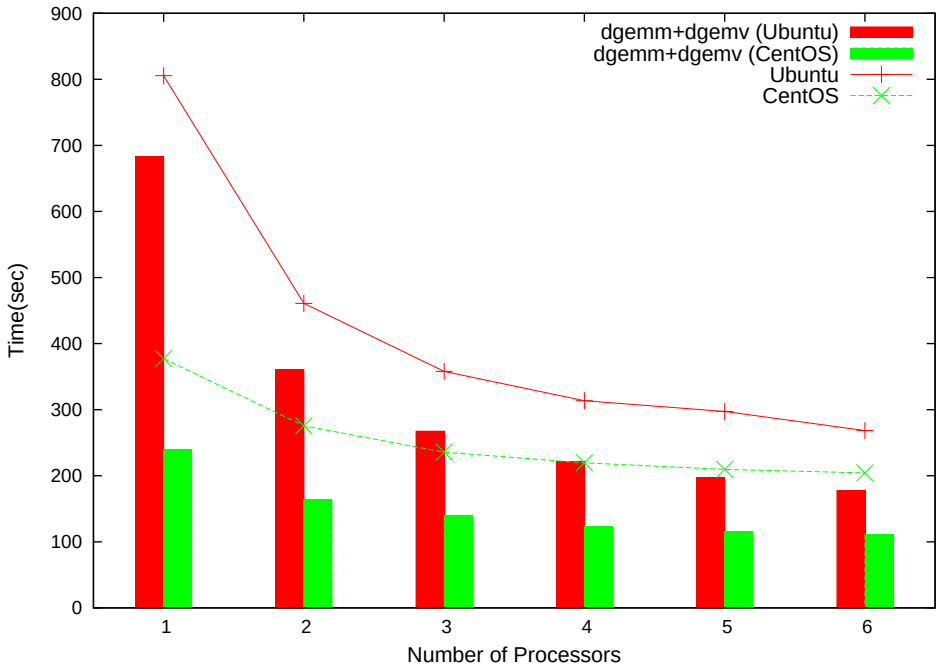

図 38: 全体の処理時間と dgemm+dgemv の処理時間を比較

CentOS の処理時間の差は，dgemm+dgemv の処理時間の差とほぼ同じである．よって，Ubuntu と CentOS の性能差は dgemm と dgemv の性能差が関わっていることが分かる．

gprof を用いた調査により，dgemm+dgemv の処理時間の差が Ubuntu と CentOS の性能差に関わっていることが分かった．この差を解消することができれば，固有値計算のさらなる高速化を図ることができる．

6　行列・ベクトル計算の最適化

調査の結果，dgemm と dgemv が性能差の原因であることが分かった．dgemm は行列と行列の積を，dgemv は行列とベクトルの積を計算する BLAS のサブルーチンである．

6.1　ATLAS, GotoBLAS とは

ATLAS はテネシー大学の Whaley らが開発したライブラリで，対象プロセッサのハードウェアリソースに適したライブラリを自動生成する [8]．GotoBLAS はテキサス大学の後藤らが開発している BSD ライセンスのオープンソースソフトウェアである．ATLAS, GotoBLAS を用いることで，BLAS のサブルーチンを最適化することができる．

6.2　仮想マシンを用いた ATLAS, GotoBLAS の性能評価

仮想マシンを用いた ATLAS, GotoBLAS の性能評価を行った．評価対象は遷移確率行列での Krylov-Schur, その際の条件は行列サイズ 720,600×720,600，最大繰り返し回数 1,000 回である．GotoBLAS は複数プロセッサ利用可能な場合，そのプロセッサ数に応じた複数スレッド版のライブラリを作成する．例えば 6 プロセッサ利用可能な場合，GotoBLAS は 6 プロセッサに最適化されたライブラリを作成する．このライブラリを用いて SLEPc による並列計算を行うと，実行時に`-np` オプションで 4 プロセッサ使用すると指定した場合，合計で 24 プロセス (6×4) が実行され，逆に性能が悪くなる．そこで，シングルスレッド版の GotoBLAS ライブラリを用い，SLEPc による並列計算を行った．

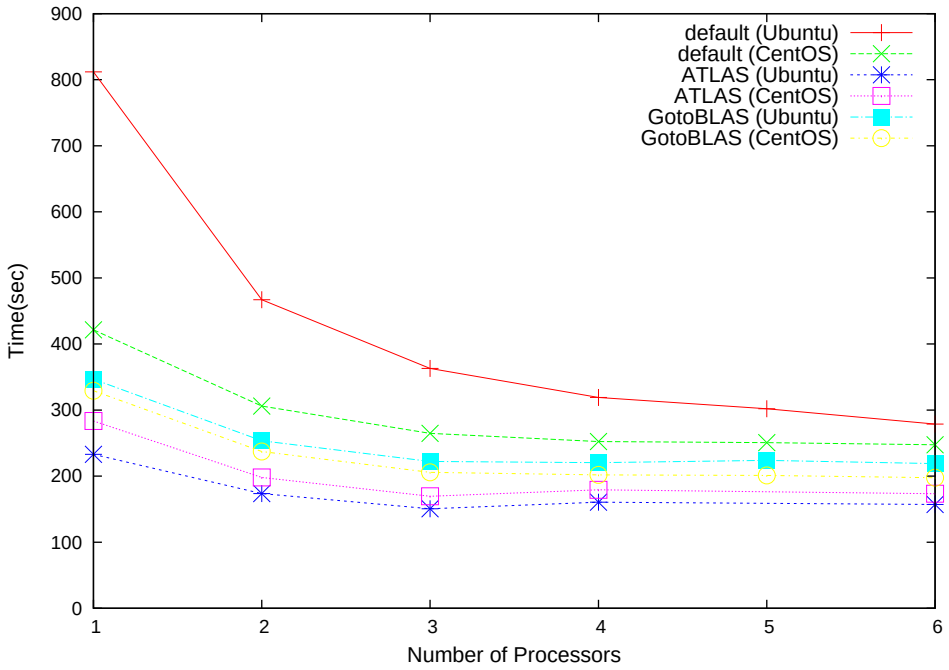

図 39: ATLAS, GotoBLAS の有無による処理時間の比較

図 39 は ATLAS, GotoBLAS の有無による固有値計算の処理時間を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間 (秒) である．また，プロセッサ数 5 のときの ATLAS の結果がないのはエラーにより処理が途中で中断してしまったためである．これより ATLAS, GotoBLAS を用いることで確かにデフォルトに比べ処理時間が短縮できていることが分かる．デフォルトでは CentOS の方が処理時間が短かったが，ATLAS, GotoBLAS では Ubuntu の方が短くなっていることが分かる．また，ATLAS と GotoBLAS を比較すると，Ubuntu, CentOS ともに ATLAS の方が良い結果となった．

6.3　ATLAS, GotoBLAS に関する考察

ATLAS, GotoBLAS を用いることで確かに処理時間を短縮することができた．ATLAS と GotoBLAS を比較すると，ATLAS の方が処理時間が短い．しかし，ATLAS ではエラーが発生し処理が中断してしうことがあっ

た．GDB を用いた調査により，MPI_Allreduce という関数が呼び出された際にセグメンテーションフォールトが発生することがエラーの原因であることが分かっている．また，このエラーの原因は Hyper-Threading や仮想マシンを用いたことが原因ではない，しかし，現段階ではこのエラーの解消はできていない．よって，現段階では ATLAS よりは遅いものの，安定に動作する GotoBLAS の利用を検討している．

6.4　agile, social への ATLAS, GotoBLAS の適用

評価環境 agile, social に ATLAS, GotoBLAS をインストールし，性能評価を行った．

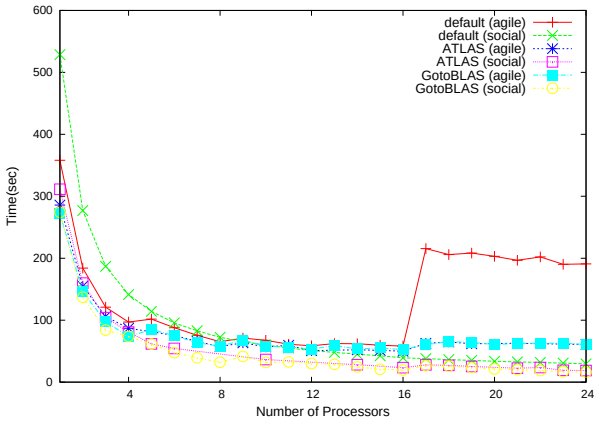

図 40: 遷移確率行列での Krylov-Schur の処理時間の比較

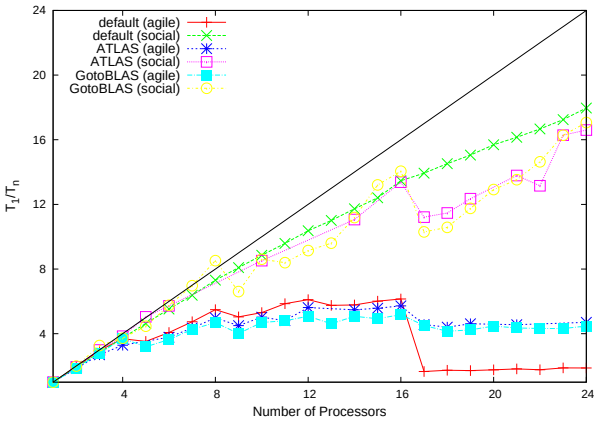

図 41: 遷移確率行列での Krylov-Schur の性能向上率の比較

物理マシン上でも ATLAS, GotoBLAS を用いた最適化の効果が現れていることが分かる．しかし，ATLAS は仮想マシンでも見られたエラーによる中断が多く見られた．また，仮想マシン上では，ATLAS と GotoBLAS を比較すると，ATLAS の方が処理時間が短かったが，物理マシン上では僅かに GotoBLAS の方が短い結果となった．これは，GotoBLAS のインストール時に利用している CPU のアーキテクチャ(agile, social ともに Nehalem) を指定したのだが，これにより，最適化が正しく働いたためでないかと考えられる．これより，処理時間が短く，安定して動作する GotoBLAS を用いるのが良いと考える．また，social の性能向上率が agile と比較するとかなり良いのは，social でのプロセッサ数 1 の場合の処理時間が極端に長いためである．また，これは ATLAS, GotoBLAS を適用した際にも見られた．

7　まとめと今後の課題

agile, social に ATLAS, GotoBLAS をインストールし，性能評価を行ったが，仮想マシンの結果のように性能差を解消しきることはできなかった．しかし，social において，遷移確率行列での Krylov-Schur の処理時間が，48 プロセッサで 20.29 秒だったものが，GotoBLAS を用いて最適化を行うことで，16 プロセッサで 19.51 秒と 1/3 のリソースで同等の性能を得ることが可能となった．これより，固有値計算の高速化には成功している．しかし，処理時間のグラフから分かる通り，プロセッサ数の増加に伴い，処理時間の短縮は

できているものの，グラフが平行線に近づきつつあることから，プロセッサを増やしてもこれ以上の短縮は見込めない．これまで，収束誤差を 10^{-7} として固有値計算を行ってきた．収束誤差を緩めることで，繰り返し回数が減少し，処理時間の短縮も見込まれる．しかし，収束誤差の調整には，実際に提案システムでユーザが必要とするサービスの品質を知る必要があるが，それには実際にランキング計算の対象となるデータを用いた試行が必要となる．

謝辞

本研究を進めるにあたり，ご指導いただいた秋山豊和准教授に感謝いたします．また，共同研究させていただいた河合由起子准教授および河合研究室の皆様には，刺激的な議論やご意見を頂き，ありがとうございました．最後に，同じ研究室の皆様には，困ったときに貴重な意見を頂きました．特に，4 回生の久保田吉徳君とは共同研究を進め，彼との議論と切磋琢磨なしには私の有意義な研究生活はあり得ませんでした．

参考文献

[1] Lawrence Page and Sergey Brin and Rajeev Motwani and Terry Winograd,“The PageRank Citation Ranking: Bringing Order to the Web”, Technical Report, No.1999-66, Stangord InfoLab (1999).

[2] Department of Astronomy, Kyoto University, Hajime BABA, Ph.D, ”Google の秘密-PageRank 徹底解説”, http://homepage2.nifty.com/babahajime/wais/pagerank.html

[3] Vicente Hernandez and Jose E.Roman and Vicente Vidal,“SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems”, ACM Tran- sctions on Mathematical Software, Vol.31, No.3 pp.351-362, Sep.2005

[4] SLEPc - Scalable Library for Eignvalue Problem Computations, http://www.grycap.upv.es/slepc/

[5] PETSc: Home Page, http://www.mcs.anl.gov/petsc/

[6] Automatically Tuned Linear Algebra Software (ATLAS), http://math-atlas.sourceforge.net/

[7] GotoBLAS - Texas Advanced Computing Center, http://www.tacc.utexas.edu/tacc-projects/gotoblas2

[8] 成瀬彰，住元真司，久門耕一，”Xeon プロセッサ向け Linpack ベンチマーク最適化手法とその評価 (性能最適化)”, 情報処理学会論文誌 コンピューティングシステム, Vol45 (2004)