コンピュータ理工学部特別研究 II A・B

オープンソースによるランキング計算と検索エンジンへの組み込み方式の検討

A consideration of ranking calculation method and its integration method into search engines using open source

秋山研究室

久保田　吉徳

平成 24 年 5 月 12 日

概要

我々の研究グループでは，Web ページの閲覧者間でのリアルタイムなコミュニケーションを実現するプロトタイプを構築し，評価を行っている．本研究では，プロトタイプシステムに，提案するランキング機能を追加するため，提案システムの構成について再検討を行った．検討の結果，システムが提案する機能を，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つの機構に分類した．プロトタイプシステムが提供する機能は，コミュニケーション機構に相当し，本研究では残るクローリング機構，ランキング機構，全文検索機構について調査を行った．ランキング機構は，固有値計算手法を用いた実装と，その他の手法を用いた実装について比較評価を行った．現時点では，固有値計算手法を用いたものが良い結果を示しているが，MapReduce を用いた手法についても今後調査する必要がある．クローリング機能および全文検索機構については，調査の結果オープンソースライブラリである，Nutch および Solr を用いることで実現できることがわかった．また，構築したランキング機構と連携するために必要な機能拡張について調査を行った．今後これらの機能を実現していく予定である．

Abstract

Our research group has implemented a prototype system that provides real-time communication functions among visitors of web pages, and the system is now under evaluation. In this study, in order to add proposed ranking function to the prototype system, we reconsidered the structure of the proposed system. As a result, the proposed system was divided into four subsystems, communication, crawling, ranking and full-text search subsystem. The functions provided by the prototype system was equivalent to the communication subsystem. So thus, we investigated the other subsystems in this study. In the ranking subsystem, we compared three implementations, one of them was based on the eigenvalue calculation. While the temporal result showed that eigenvalue implementation was the best, MapReduce implementation was not tested yet and it should also be examined. Regarding crawling and full-text search subsystem, as a result of the investingation, Nutch and Solr, open source libraries, could satisfy our requirements. We also inverstigated how to integrate our ranking subsystem with them. The implementation of the integration is the future work.

目 次

1　はじめに

近年，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，自分の知りたい情報を検索する際に，大量の Web ページが表示されるため，検索が困難になっている．Web ページの検索を行う場合，２つの方法が考えられる．一つは，Yahoo!や Google などの検索サービスの利用であり，もう一つは mixi や２チャンネルなどの掲示板や SNS の利用である．検索サービスは速度と網羅性の点で有効である．掲示板や SNS による情報収集は検索サービスよりも時間や手間を要するが，コミュニケーションにより質の高い情報が得られるという利点がある．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能，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を実現し，検索とコミュニケーションの融合を目指す．提案システムの実現により，閲覧者が欲しい Web ページの情報に加えて知識を豊富に持つユーザからの情報を同時に獲得できる可能性がある．既に閲覧者のリアルタイムなコミュニケーションを実現するプロトタイプシステムを構築し，評価を行っている．本研究では，既存のプロトタイプシステムにランキング機能を追加するため，ソーシャルサーチシステムの構成を再設計し，ランキング機能の追加方法を検討する．また，ランキング機能の実装とその機能の評価，検索エンジンへの組み込み方法について検討する．以下，2 章では，ソーシャルサーチシステムの設計について述べる．3 章では，ランキング計算手法について述べ，固有値計算を用いた手法とその他の手法について比較する．4 章では，オープンソースの検索エンジンについての調査結果を示し，ランク値の組み込み方法について検討する．最後に 5 章でまとめおよびこれからの課題について述べる．

2　ソーシャルサーチシステムの設計

これまでに，ソーシャルサーチシステムのプロトタイプを実装している [1]．プロトタイプシステムは，閲覧者数に応じた検索結果のリランキング機能および，Web ページ上での閲覧者同士のコミュニケーション機能を提供している．さらにインターフェイスを改良したシステム，” ぺちゃくちゃ検索” を開発し，goo ラボで試験的にサービスを公開している [2]．プロトタイプシステムでは，goo や Yahoo!などの検索エンジンの API を用いて検索機能を実現していたため，ページ間のリンク構造の情報が参照できず，提案するランキング機構が実現できていない．そこで本研究では，プロトタイプシステムに提案するランキング機能を追加するため，まずソーシャルサーチシステムのシステム構成について再検討した．

閲覧者の量と質に基づいたランキング，および，ページを通して閲覧者とリアルタイムなコミュニケーションを提供するソーシャルサーチシステムを実現するためには，以下のような機能を実現する必要がある．

(a) 閲覧者間のリアルタイムコミュニケーション機能

(b) ブラウザの拡張機能を用いてリアルタイムに閲覧履歴など Web ページ閲覧者の情報を収集する機能

(c) 閲覧者のプロファイルや閲覧者間の関係を抽出する機能

(d) Web ページをクローリングしてリンク関係を抽出し，インデックスを生成する機能

(e) 抽出した情報からランク値を計算する機能

(f) クローリングしたドキュメントの全文検索機能

(g) ランク値を検索結果に反映する機能

本研究では，これらの機能を，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つの機構に分類する．コミュニケーション機構では，(a),(b),(c) の機能を実現する．閲覧者間の関係の抽出には，Facebook や Twitter などのソーシャルサービスを利用する．クローリング機構では，(d) の機能を実現する．ランキング機構では，(e) の機能を実現し，コミュニケーション機構および，クローリング機構により収集した情報に基づいて隣接関係を作成し，PageRank[3] によりランク値計算を行う．そして，全文検索機構では，クローリングしたドキュメントおよび閲覧者情報について，(f),(g) の機能を実現し，より質の高い検索結果を返す．この４機構の関係を図 1 に示す．

以下 3 章では，ランキング機構でのランキング計算手法に関する調査，実装および評価について述べる．4 章では，クローリング機構および全文検索機構との連携について述べる．

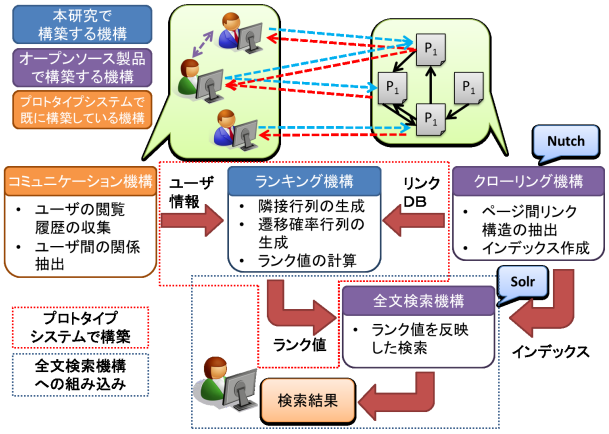

図 1: ソーシャルサーチシステムの構成図

3　ランキング機構の実装と性能評価

3.1　ランキング計算の概要

ランキング機構では，コミュニケーション機構および，クローリング機構により収集した情報を PageRank に基づいて隣接行列を生成し，ランキング計算を行う．まず PageRank とは何かを述べる．

3.1.1　PageRank とは

PageRank は文献 [3] で提案され，ページの重要度の自動判定技術であり，Web ページ [4] 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したもので，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．

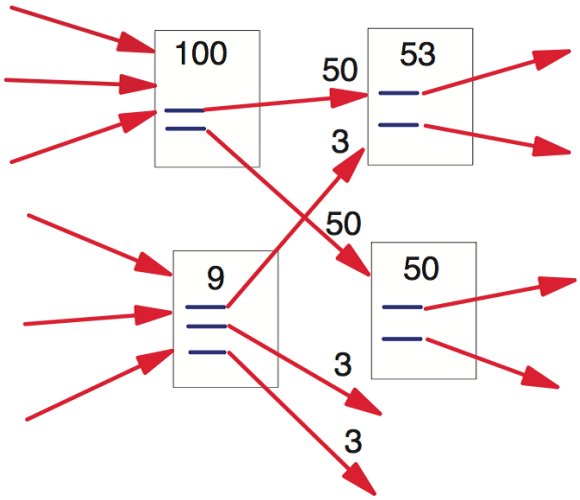

図 2: PageRank の考え方 ([3] から引用)

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 [3]．” 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 ページに与える 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回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 [3].

3.1.2　行列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 として，行列を生成する．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行列を使用し，この行列を” 遷移確率行列” という．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．

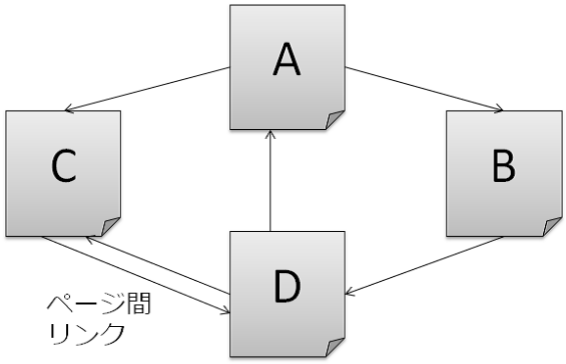

図 3: Web ページのリンク構造

Web ページのリンク構造における行列生成の例

図 3 のようにページ A,B,C,D のリンク構造での隣接行列および遷移確率行列の生成方法について説明する．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 がたちます．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，下のような隣接行列になります．

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

次に隣接行列から遷移確率行列を生成します．隣接行列は隣接行列の列成分をリンク数で割ったものになるので，1 列目はリンク B,C の 2 つですので，各成分を 2 で割り，行列の (2,1),(3,1) に $\frac{1}{2}$ をたてます．同様に，他の列成分もリンク数で割り，下のような遷移確率行列になる．

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

このようにして求めた遷移確率行列の最大固有値を求めることにより，ランク値を求めることができ，図 3 のランク値は $[A, B, C, D] = [0.365, 0.182, 0.574, 0.730]$ となる．

ソーシャルサーチシステムにおける行列生成の例

ソーシャルサーチシステムでは図 3 の各ページの閲覧者を考慮し，” 各ユーザが他の閲覧者を介して別のページに遷移する” というモデルを考える．まずは単純に，4 のように閲覧者から閲覧しているページへの

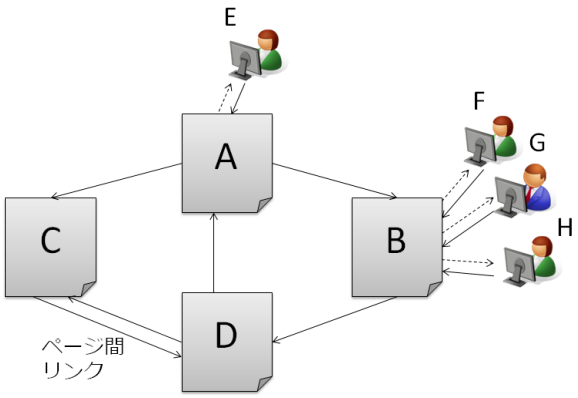

図 4: Web ページのリンク構造と閲覧者

リンクとその逆方向のリンクを追加するモデルを考える．このとき，隣接行列および遷移確率行列は以下になる．

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{array}
\begin{array}{c}
\begin{array}{cccccccc} A & B & C & D & E & F & G & H \end{array} \\
\left[\begin{array}{cccc|cccc}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\end{array}
\qquad
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{array}
\begin{array}{c}
\begin{array}{cccccccc} A & B & C & D & E & F & G & H \end{array} \\
\left[\begin{array}{cccc|cccc}
0 & 0 & 0 & \frac{1}{2} & 1 & 0 & 0 & 0 \\
\frac{1}{3} & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\frac{1}{3} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\
0 & \frac{1}{4} & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline
\frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
\end{array}
$$

この遷移確率行列よりランク値計算すると，$[A, B, C, D, E, F, G, H] = [0.408, 0.544, 0.408, 0.544, 0.136, 0.136, 0.136, 0.136]$ となり，閲覧者数が一番多いページ A,B の優先度が高くなっているのがわかる．すなわち，現在注目されているページが優先されることになる．単純に閲覧しているページをリンクしただけでは，閲覧者を経由した遷移確率は低くなるが，例えば，閲覧者が過去に参照したページ，閲覧者がブックマークしているページ，閲覧者自身のブログページなども関連が深い可能性があり，リンクを追加することで閲覧者から得られる情報を検索に反映できる可能性がある．ここで興味深いのは，閲覧者自身にもランク値が付けられることであり，ユーザ自身の検索内容に近い興味をもつユーザの検索に，このランク値が適用できる可能性がある．また，SNS 等から抽出される閲覧者同士の関係を用いる方法も考えられる．これにより，疎になっている遷移確率行列の右部分にも関連が生じ，コミュニティから得られる情報をより反映できる可能性がある．

閲覧者同士の関係性などを抽出するコミュニケーション機構において考慮する必要があるが，本研究では，ランキング機構における，ランク値計算について調査し，求めたランク値を検索エンジンへの組み込む手法を検討する．ランク値計算手法については，3.2〜3.5 節で，ランク値の検索エンジンへの組み込み手法については 4 章で述べる．

3.1.3　グラフを用いたランキング計算

その他のランキング計算として，PageRank 計算を実装しているオープンソースライブラリ JUNG[7] がある．JUNG は，Java で記述されており，グラフで表現できる情報の解析や視覚化を行うためのフレームワークで，有向グラフや無向グラフ，マルチモーダルグラフなどを扱うことができる．また，今回調査する PageRank 計算のようなグラフ理論やデータマイニング，ソーシャルネットワーク解析から様々なアルゴリズムが実装されている．他にも，Hadoop の MapReduce の分散処理を用いた計算手法提案されている [9]．以下それぞれの手法について説明する．

JUNG を用いたランキング計算

JUNG の計算手法について，Eclipse のデバック機能を用いて調査を行った．JUNG の PageRank 計算は以下の繰り返しにより算出されている．

$$\begin{cases} \sum_{m \in G} P(n) = 1 \\ P(n) = \frac{1}{|G|} \end{cases} \tag{1}$$

$$P(n) = \alpha \ (\frac{1}{|G|}) + (1 - \alpha)(\sum_{m \in L(n)} (IW(n,m) \times P(m))) \tag{2}$$

ここで，P が PageRank 値，—G—が PageRank を求める対象ページの総数，L(n) がページ n にリンクしているページの総数，が IW(n,m) がページ m からページ n へのリンクウエイト，αがジャンプ係数である．式 (1) に示すように P(n) の初期値はすべてのページに同じ PageRank 値を与え，全ノードの P(n) の和は必ず 1 になる．繰り返し行われる計算式 (2) は文献 [4] または文献 [8] の 3.2 節で述べられているべき乗法と同じ計算である．第 1 項はすべてのノードからのランダムジャンプによるインリンクの影響を示している．

MapReduce を用いたランキング計算

文献 [9] では．式 (2) の PageRank 計算をを Hadoop の MapReduce を用いて，分散処理する方法を紹介している．Hadoop とは，大規模データの分散処理を支える Java ソフトウェアフレームワークである．図 3 の例を用いて MapReduce による PageRank 計算について述べる．もともと，PageRank の計算は，各ページのランク値をアウトリンク先に均等に分け，各ページの新しいランク値はインリンクから寄せられたランク値の合計がそのページのランク値になる (3.1.1 節で説明)．Map および Reduce のイメージを図 5,6 に示す．

MapReduce では，ランク値を均等に分ける操作を Map の操作，ランク値を合計する操作を Reduce の操作とする．はじめ，各ページには均等に PageRank 値を与え，そこから MapReduce の操作を行う．図 5 では，P(A),P(B),P(C),P(D) の初期値を $\frac{1}{4}$ とする．次にページ A はページ B,C にリンクしているので，P(A) をページ B,C に均等に $\frac{P(A)}{2}$ ずつ与える．他のページも同じようにリンク数で割った値をリンク先ページに与える．6 では，map 操作により与えられた PageRank 値をそれぞれのページごとに集め，それを足すことにより新たな PageRank 値がページに与えられる．この操作を繰り返し続けることにより，値は収束して，Map する前と Reduce した後の値が変化しなくなり，この値が PageRank 値となる．しかし，この操作には JUNG に実装されていたランダムジャンプ係数を考慮していないため，正しい PageRank 値とはいえない．

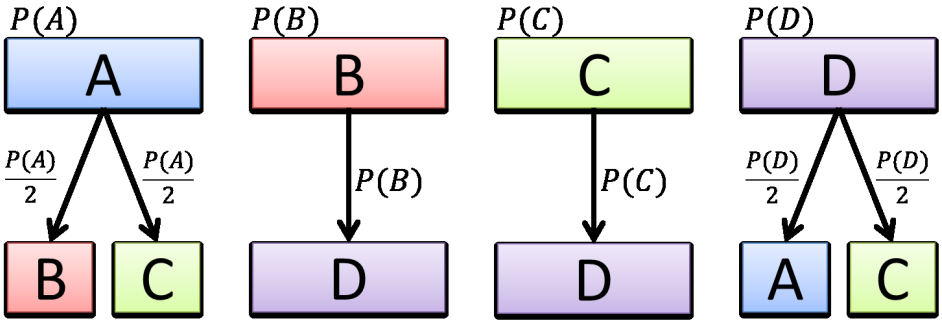

図 5: map の操作

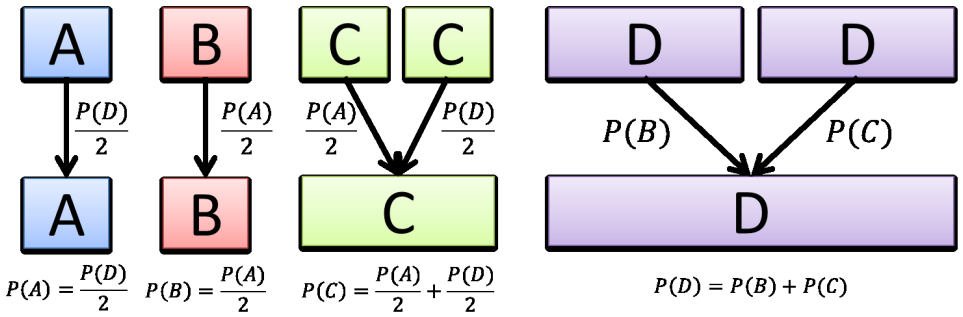

図 6: reduce の操作

そのため，MapRedece 操作した後にランダムジャンプ係数を考慮する必要があり，ランダムにジャンプする確率を α，ジャンプした時あるページにいる確率は $\frac{1}{|G|}$(G は PageRank を求めるページの総数) で表すと，以下の式により求められる．

$$P(n) = \alpha\ (\frac{1}{|G|}) + (1 - \alpha\ ) \sum_{m \in L(n)} (\frac{P(m)}{C(m)}) \tag{3}$$

ここで，P は PageRank，C はアウトリンク数，L(n) はページ n にリンクしているページの総数である．この式 (3) は式 (2) と同じであることがわかり，JUNG と同じように式 (1) も満たす．このようにして計算を行うことによって，初期に与えられた PageRank 値の合計が，更新された PageRank 値の合計と一致する．しかし，図 3 と違い，ページのリンク構造の中には，外部リンクのないページも多数存在する．文献 [9] によると，外部リンクのないページがある場合，MapReduce 操作の前後で PageRank の合計値が一定に保存されないため，これを考慮する必要がある．この考慮の方法の一つとして，MapReduce 後に失われた PageRank 値をすべてのページに均等に分けることで，PageRank の合計値を一定に保存することができる．この 2 つの動作を繰り返し行うことにより，PageRank 値は一定の値に収束する．

3.2　固有値計算の高速化手法

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きたため，これまでに様々な方法が提案されてきている．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り，LR 法の計算手順は以下である．

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} は右上三角行列に収束し，その対角成分が A の固有値となる．

QR 法は行列 A をユニタリ行列 Q と右上三角行列 R に分解する手法であり，LR と違い，非対称行列の場合にも計算が可能である．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・シュミット直交化に基づいて行う．以下が分解手順および QR 法の計算方法である．

グラム・シュミット直交化法に基づいた QR 分解の計算方法

1. 行列の列成分に着目し，行列 A の 1 列目を $\mathbf{a_1}$ とすると $\mathbf{b_1} = \mathbf{a_1}$ を基底に設定し，正規化[a]を行い $\mathbf{q_1}$ を生成する．
   $\mathbf{q_1} = \frac{\mathbf{b_1}}{|\mathbf{b_1}|}$
2. 行列 A の二列目 $\mathbf{a_2}$ を用いて q_1 と直行な基底 $\mathbf{b_2}$ を生成する．$\mathbf{b_2} = \mathbf{a_2} - (\mathbf{q_1}, \mathbf{a_2})\mathbf{q_1}$
3. $\mathbf{b_2}$ を正規化して $\mathbf{q_2}$ を生成する．
   $\mathbf{q_2} = \frac{\mathbf{b_2}}{|\mathbf{b_2}|}$
4. これを繰り返し行い，$\mathbf{q_1}$〜$\mathbf{q_n}$ 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 $r_{ij} = |b_i|(i = j)$ で求める．

[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} は右上三角行列に収束し，その対角成分が A の固有値となる．

しかし，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．射影法は，行列が対称行列の場合は，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相似変換し，QR 法に基づいて計算を行うことで，計算時間が削減できる．射影法を用いて計算を行っている Krylov-Schur 法は，Arnoldi の原理に従ってベクトル列とヘッセンベルグ行列分解し，得られた行列をさらに QR 法に基づいて計算することで固有値を求める．また，固有値が求まった部分からロックし，Arnoldi 分解および QR 法を反復試行させて固有値を求める．このように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アプリケーションでは，大幅に計算時間を削減できる可能性がある．Arnoldi の原理とは，正則な行列 A と非ゼロベクトル u_1 から正値エルミート行列の正規直交系を作ることである．Arnoldi の原理によって算出される式および Krylov-Schur 法の計算を以下に示す．

Arnoldi の原理によって算出される式

1. 行列 A と非ゼロベクトル u_1 によって作られたベクトル列に対してグラム・シュミット直交化に基づいてヘッセンベルグ行列 H_m とベクトル列 V_m に分解する
   $AV_m = V_m H_m + h_{n+1} v_{n+1} e_n^T$
2. 求めた式を移項して $AV_m - V_m H_m = h_{n+1,n} v_{n+1} e_n^T$ とするとき，$h_{n+1,n} v_{n+1} e_n^T = f$ とし，f を Arnoldi の残差という

Krylov-Schur 法

1. Arnoldi 分解し，得られた H_m を B_m とする．$AV_m = V_m B_m + v_{m+1} b_{m+1}^T$
2. B_m に対して QR 分解を施し，T_m を作る．$B_m = Q_1 R_1$
   $T_m = Q_1^{-1} B_m Q_1$
3. これより，元の式は以下のように表すことができる．$AV_m Q_1 = V_m Q_1 Q_1^{-1} B_m Q_1 + v_{m+1} b_{m+1}^T Q_1$
   $AV_m Q_1 = V_m Q_1 T_m + v_{m+1} b_{m+1}^T Q_1$
4. 固有値の求まったところからロックし，求まってないところを再び Arnoldi 分解，QR 分解を施し，反復させる．

他に射影法を用いて計算をおこなっている Lanzcos 法は，行列が対称行列の時にのみ計算が可能であり，クリロフ部分空間に射影することにより，三重対角行列に相似変換し固有値を求める手法がある．また，Arnoldi 法は非対称行列の時に有効であり，クリロフ部分空間に射影することにより，ヘッセンベルグ行列に相似変換し．固有値を求める手法である．本研究では，固有値計算ライブラリを実装している SLEPc と PageRank 計算をサポートするライブラリを実装している JUNG について比較した．

3.3 固有値計算ライブラリ SLEPc を用いたランキング計算の実装

SLEPc(Scalable Librart for Eigencalue Problem Computations) [5] 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 [6], 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．なお，SLEPc および PETSc のインストール方法，使い方は付録の 6.1 節，使用できるコマンドについては 6.2 節を参照してほしい．

3.4 SLEPc の性能評価

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．

評価に用いたのは評価環境 agile(表 1) である．

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 7(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 7(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．

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

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

(a) ラプラシアン行列　　(b) 遷移確率行列

図 7: 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．以下の実験では許容誤差を 10^{-7} として測定を行った．

3.4.1 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の固有値計算にかかる時間を測定した．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 8 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

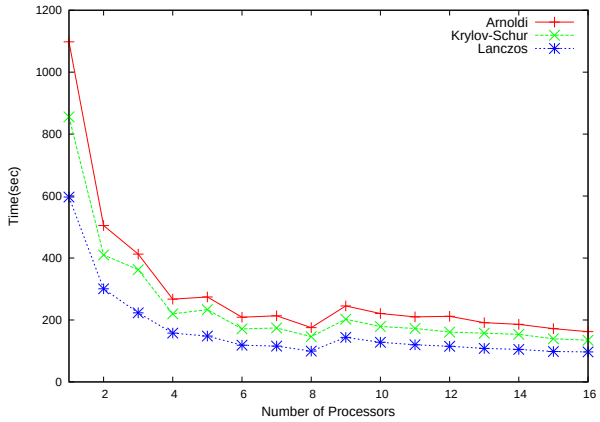

図 8: 単一ノードでのラプラシアン行列における処理時間

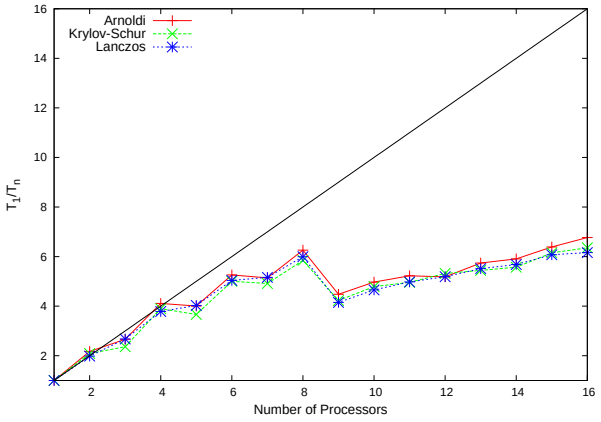

図 9: 単一ノードでのラプラシアン行列における処理時間比 (T_1/T_n)

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．

次に単一プロセッサで計算したときの処理時間を T_1，プロセッサ数 n で計算をしたときの処理時間を T_n としたとき，プロセッサ数を変化させた場合の処理時間比 T_1/T_n を比較したものを図 9 に示す．ここで，横軸はプロセッサ数，縦軸は処理時間比 (T_1/T_n) である．図 9 の $y = x$ の直線は理想的な性能を表したものであるが，実際に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処理時間比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，処理時間比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．

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 P は，

$$P(n) = \frac{T_1}{T_n} = \frac{T_1}{RT_1/n + (1-R)T_1} = \frac{n}{R + n(1-R)} \tag{4}$$

となる．

これにより，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．

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 (4) 式から求めると，Arnoldi では 0.9675，Krylov-Schur では 0.9491，Lanczos では 0.9582 となり，Arnoldi が最も良い数値を示した．しかし，これはプロセッサ数 8 までの結果であり，プロセッサ数 9 以降の結果を含めると並列化率は低下すると考えられる．

3.4.2 Hyper-Threding 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 8, 9 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-Threding 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，Hyper-Threading を無効にして再度測定を行い，Hyper-Threading の有無による処理時間比 (T_1/T_n) を比較した．その結果が図 10 である．ここで，横軸はプロセッサ数，縦軸は処理時間比である．

どの計算手法においても，Hyper-Threading が無効の結果の方が良いことが分かる．また，有効の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考えられる．

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，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．

3.4.3 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 7(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 11 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

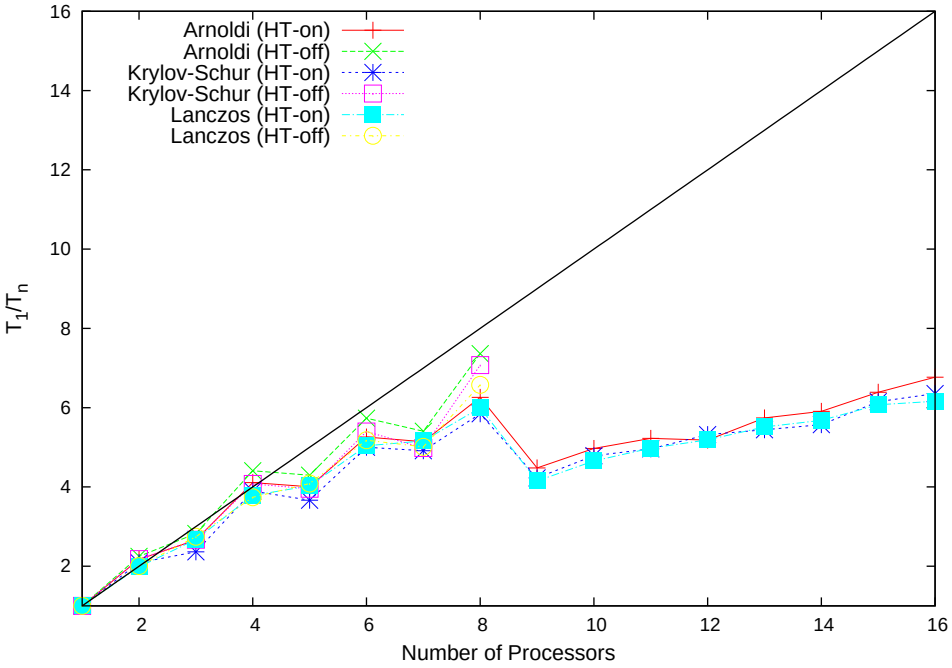

図 10: Hyper-Threading の有無による処理時間比 (T_1/T_n) の比較

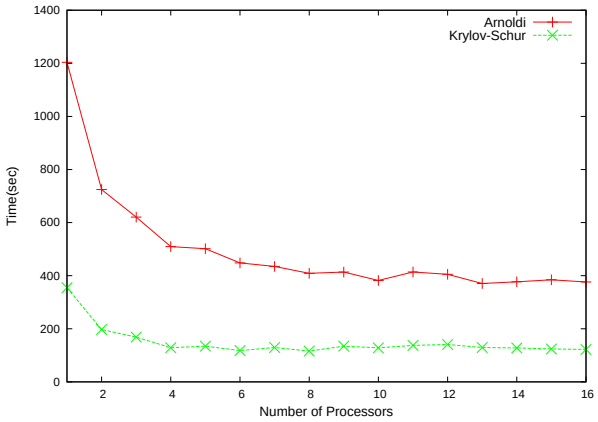

図 11: 単一ノードでの遷移確率行列における処理時間

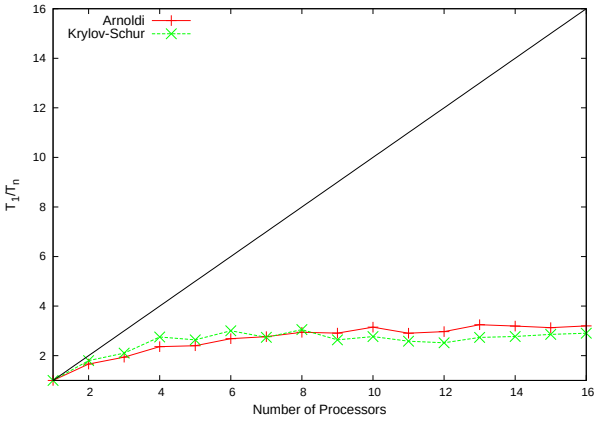

図 12: 単一ノードでの遷移確率行列における処理時間比 (T_1/T_n)

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 のときの両者の処理時間の差は，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 12 は単一ノードでの遷移確率行列における処理時間比 (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，縦軸は処理時間比である．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 (4) 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 Hyper-Threading が原因と思われる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．

3.4.4 複数ノードでのラプラシアン行列の性能評価

次に，複数ノードを用いた際の TCP/IP による影響を調査した．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，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った．プロセッサ数としては合計で 24 プロセッサ利用可能である．図 13 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．

図 14 は複数ノードでのラプラシアン行列における処理時間比を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間比である．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が，これは単一ノードの場合と同じように Hyper-Threading によるものであると考えられる．また，単一ノードでは処理時間比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 (4) 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．

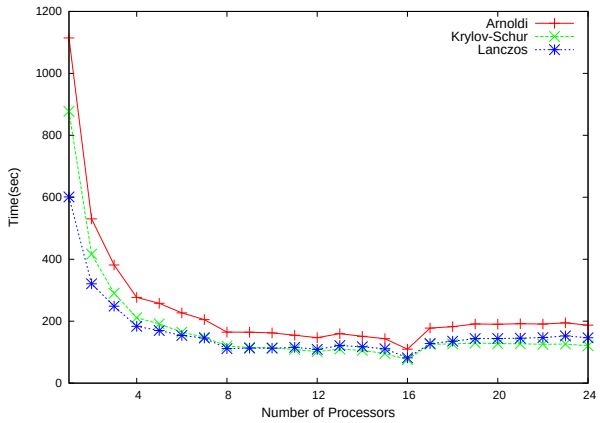

図 13: 複数ノードでのラプラシアン行列における処理時間

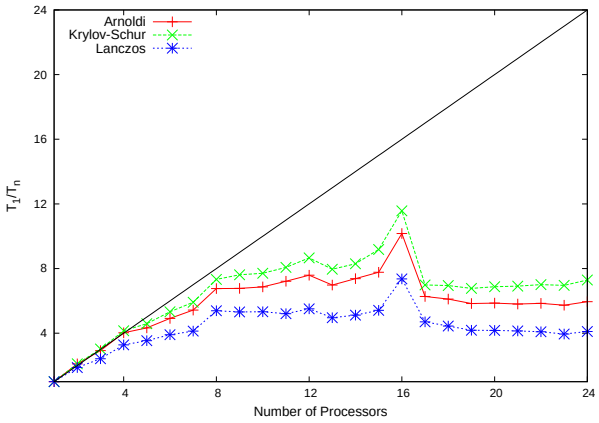

図 14: 複数ノードでのラプラシアン行列における処理時間比 (T_1/T_n)

3.4.5 複数ノードでの遷移確率行列の性能評価

図 15 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．単一ノードでは Hyper-Threading による極点な性能低下は見られなかったが，複数ノードではプロセッサ数 17 以降の性能が極端に悪くなっている．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．Hyper-Threding による性能低下を含めても Krylov-Schur は Arnoldi に比べかなり性能が良いことが分かる．

図 16 は複数ノードでの遷移確率行列における処理時間比を比較したものである．ここで，横軸はプロセッサ数，縦軸は処理時間比である．単一ノードでは Arnoldi と Krylov-Schur の処理時間比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 のほう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 (4) 式から求めると，Arnoldi では 0.8368, Krylov-Schur では 0.9195 となった．

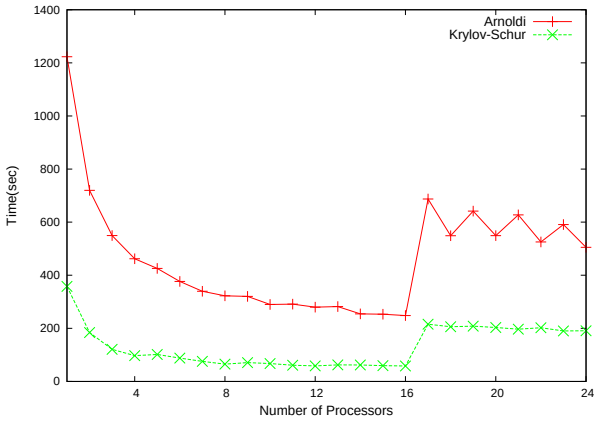

図 15: 複数ノードでの遷移確率行列における処理時間

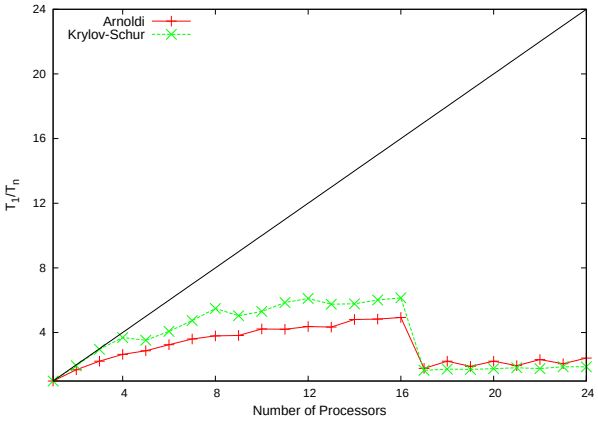

図 16: 複数ノードでの遷移確率行列における処理時間比 (T_1/T_n)

3.4.6 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通信によるオーバヘッドが考えられる複数ノードの方が処理時間がその分長くなると考えていた．しかし，図 17 よりわずかにであるが，通信によるオーバヘッドが考えられる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．

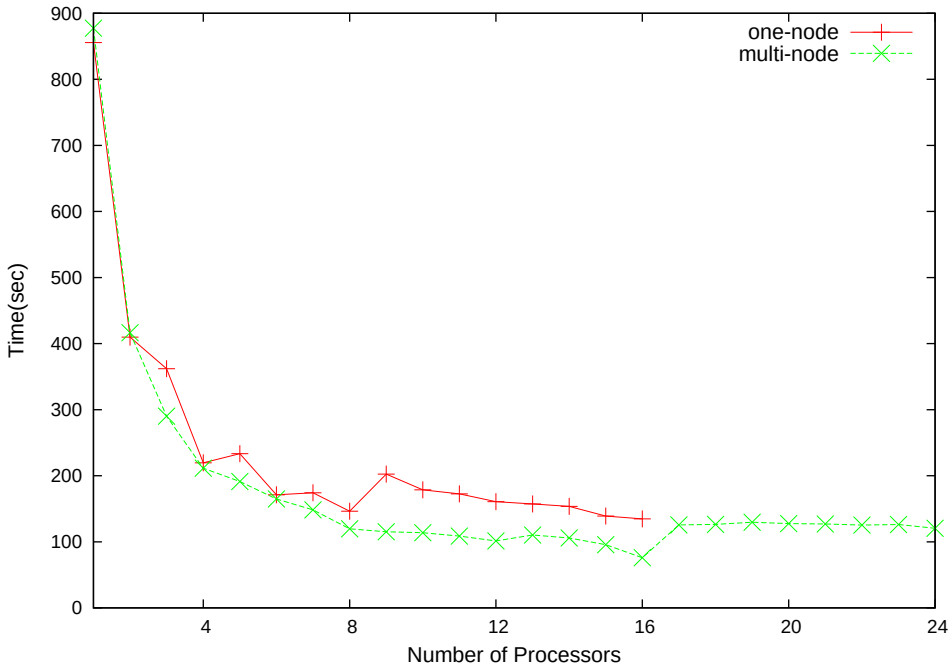

図 17: ラプラシアン行列での Krylov-Schur における単一ノードと複数ノードの処理時間の比較

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．

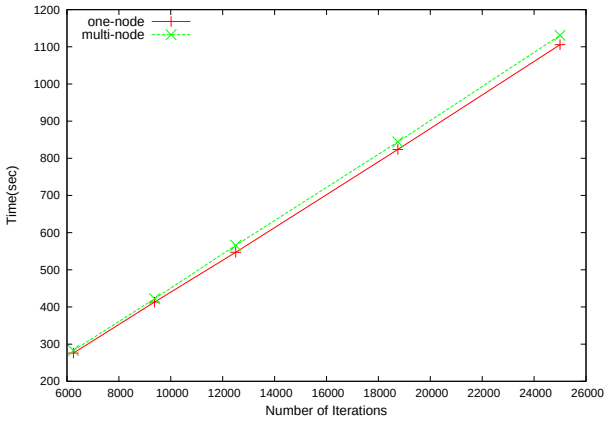

図 18: 2 プロセッサ使用したときの通信によるオーバヘッドの影響

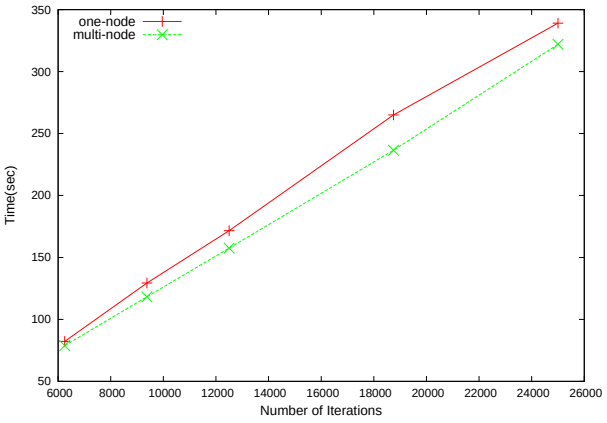

図 19: 8 プロセッサ使用したときの通信によるオーバヘッドの影響

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 (回)，縦軸は処理時間 (秒) である．

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，通信によるオーバヘッドの影響より共有メモリの待ちの影響の方が大きくなり，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．

3.5　その他のランキング計算手法との比較

3.4 節で評価した SLEPc と PageRank 計算をサポートするオープンソースライブラリ JUNG の比較を行った (JUNG の使用方法は付録 6.3 節参照)．SLEPc では，3.4 節で評価を行った通り，行列サイズが $720,600 \times 720,600$ の三角格子状をランダムに動くことを想定したモデルで，プロセッサ数を 1 で繰り返し回数を 1000 に設定して計算を行った場合，約 430 秒かかり，プロセッサ数を 8 にした場合は約 120 秒程で計算が可能である．それに対して，JUNG では，ノード数を 720,600，リンク構造を三角格子状をランダムに動くことを想定したモデルで，プロセッサ数 1 で繰り返し回数を 100 に設定した．また，JUNG ではランダムジャンプ係数を指定することができ，これは，閲覧者が必ずしもページ上のリンクをランダムに移動するだけでなく，ページ上のリンクを無視して，全く関係のないページへジャンプすることであり，PageRank の論文 [3] によると，0.15 に設定されているため，調査実験でも 0.15 に設定した．この環境において計算を行ったところ，計算時間は約 250 秒かかった．同じプロセッサ 1 で計算時間が SLEPc の $\frac{1}{10}$ にも関わらず，計算時間は SLEPc の $\frac{1}{2}$ にも及ばなかった．JUNG の性能が悪い原因について，Eclipse を用いて調査した．デバックの結果，SLEPc では，最新の固有値計算法を実装し，並列計算処理されているが，JUNG の PageRank 計算では，計算の並列化や高速化がされておらず，プロセッサ数も 1 つしか使用していなかったため性能が悪く，計算時間に多大なる時間を要したと考えられる．一方，Hadoop の MapReduce については，まだ性能評価は行えていない．文献 [9] によると，3 億 2200 万のリンクを持つグラフは，52 回の繰り返しで収束したと述べられている．この速さで，収束が可能であれば，SLEPc に比べて高速に計算する可能性があり，Hadoop の MapReduce について，評価実験を行う必要があると考えられる．

4　クローリング機構および全文検索機構との連携

本研究では，オープンソースの検索エンジンとして，Java で開発されている Nutch および Solr を用いる．Nutch は全文検索エンジンライブラリ Lucene のサブプロジェクトとして開発されており，ページのクローリングおよびスコアリングを行い，インデックスの作成を行う．Solr も Nutch 同様，全文検索エンジンライブラリ Lucene のサブプロジェクトとして開発されており，Lucene をベースに，管理画面やキャッシュ機構を取り入れたアプリケーションである．以下の節では，Nutch におけるクローリングの概要，Solr の全文検索の概要を述べた後，Nutch と Solr の連携方法を示し，検索エンジン Solr へのランキング値の反映方法に

ついて検討する．なお，Nutch のインストール方法および使い方は，付録の 6.4 節を Solr のインストール，設定方法は 6.5 節を参照してほしい．

4.1 Nutch のクローリングの概要

クローリングとは，インターネット上の Web ページのリンクをたどりながら情報を収集することである．クローリングの仕組みは，inject, generate, fetch, updatedb, invertlinks, index の 6 つのステージでクローリングを実行する．Nutch では，この一連の動作を自動的に行うが，各ステージがどのような動作を行っているかを調べるため，各ステージの動作を一つずつ実行した．

1. inject 操作

   まずクロールしたい URL の開始地点の一覧をファイルに用意しておく．そして，inject 操作を行うことにより，用意した URL 一覧を読み込み，クロールするためのデータベース (crawldb) を生成する．

2. generate 操作

   generate 操作を行うことにより，crawldb を読み込み，クロールすべき URL の抽出を行う．抽出したデータは，segments ディレクトリの配下に crawl_gnerate ディレクトリが生成され保存される．抽出する URL 数が多い場合はオプションで *topN* を指定することにより，制限することが可能である．

3. fetch 操作

   fetch 操作では．generate 操作により抽出したデータの fetch を行い，URL のコンテンツを取得し，コンテンツからリンクしている URL の抽出を行うことがでる．また，segments ディレクトリの配下に crawl_fetch ディレクトリが生成され保存される．

4. updatedb 操作

   最後に fetch により取得した URL を parse し，まだ fetch 操作が行われていない URL を取得する．取得した URL を updatedb により，crawldb に書き込みをし，再び genereta 操作をすることにより，新たな URL を fetch することができる．取得した URL がなくならない限り繰り返し実行されるが，コマンドオプション”-depth”を指定することにより制限することが可能である．

5. invertlinks 操作

   fetch する URL がなくなった，または，指定された深さまで fetch が行った後に，invertlinks 操作が実行される．この操作を行うことにより，segments ディレクトリに保存されている情報からリンク情報を取得し，linkdb を生成する．

6. index 操作

   index 操作では，1〜5 の操作で生成した crawldb, segments, linkdb の情報を用いることにより，全文検索用の index を生成する．

生成した crawldb, segments, linkdb の 3 つのデータは，コマンドを用いることにより，中身を確認することが可能であり (6.4 参照)，crawldb では，クローラが取得した URL の fetch 情報を格納しており，いつど

の URL を fetch したのか，次にクロールする時にはどの URL が fetch する必要があるのかなどが格納されている．segments では，クロールした結果が格納されており，URL の本文，タイトル，リンク，テキスト部分の抽出結果が格納されている．linkdb では，指定した URL がどこからリンクされているのかを知ることができる．また，今回ランキング計算で必要であるページのリンク構造はこの linkdb より抽出可能であり，このデータベースを解析することにより，行列の生成が可能であると考える．

4.2 Solr の検索の概要

Solr で検索を行うには，まず Solr のサーバを立ち上げる必要がある．立ち上げたサーバにブラウザでアクセスし，HTTP にて検索のリクエストを送信する．Solr は検索リクエストを受け取り，受け取った検索式を元にインデックスを作成し，結果を XML や JSON と行った形式に整形してレスポンスを返す．ここで，Solr にはサーチコンポーネントという処理単位が存在し，検索に必要となるさまざな処理がサーチコンポーネントとして実装されている．サーチハンドラは適切なサーチのコンポーネントに処理を依頼し検索を継続する．

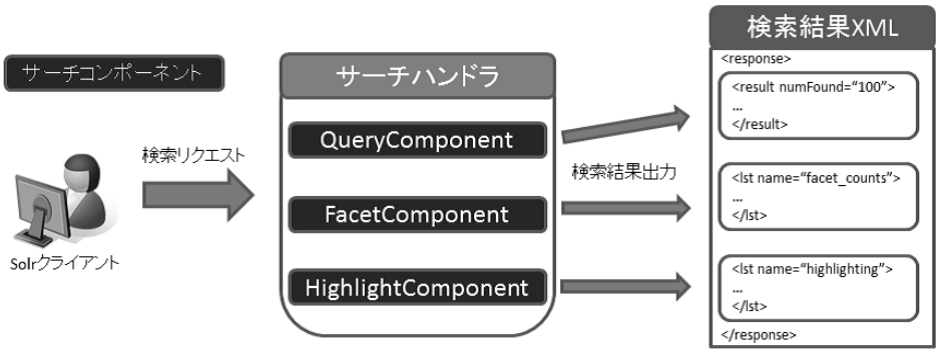

図 20: サーチハンドラとサーチコンポーネントと出力 (文献 [10] より引用)

検索式では，全文検索や単語による検索，フィールド指定検索など，様々な検索が可能である．また検索はサーチハンドラによって行われており，solrconfig.xml を設定することにより，サーチコンポーネントの追加などができる．レスポンスライタでは，XML 形式だけでなく，JSON 形式や PHP, ruby など様々な形式にサポートしている．そして，検索結果はスコアによるソートが行われている．計算式は以下である．

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t in q} (tf(t\ in\ d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d)) \qquad (5)$$

ここで $tf(t\ in\ d)$ は文書 (document:d)，含まれる単語 (term:t) の頻度係数，$idf(t)$ は文章に含まれる単語頻度を反転，$coord(q,d)$ は指定された文書で見つかった検索リクエストに含まれる単語 (検索語) の数に基づいたスコア係数，queryNorm(q) は検索リクエスト間のスコアを比較可能にするために用いる正規化係数，$t.getBoost()$ はフィールドの重要度，$norm(t,d)$ はいくつかのインデックス化時間および長さ係数をカプセル化したものである．この計算式により，計算式 q に対する文書 d のスコアが計算される．ここで，$norm(t,d)$

の中には，文書の重要度が含まれており，この値にランク値を反映することにより，提案するスコア値が得られる可能性があると考えられる．標準では，このスコアの高い順に検索結果が表示されるが，パラメータをいじることにより，降順にソートすることや，ランダムな順序でソートすることが可能である．

4.3 検索エンジンへの組み込みの検討

Nutch と Solr は連携させることができ，Nutch でクローリングした結果を Solr で表示することが可能である．連携方法は，Nutch の以下のコマンドを実行することにより，Solr に対応した形式でインデックスを生成し，インデックスおよび文書を Solr に追加する．

Solr の設定および起動方法

1. `Solr` を起動させる
   ```
   $java -jar $APACHE_SOLR_HOME/example/start.jar
   ```
2. `Nutch` により `Solr` のインデックスを生成する
   ```
   $$NUTCH_RANTIME_HOME/bin/nutch solrindex http://localhost:8983/solr
    crawl/crawldb crawl/segments/*
   ```
3. `http://localhost:8983/solr/admin` で検索を行う

ドキュメントの追加が完了すると，Solr の検索式にてクローリングした結果を検索することが可能になる．一方，ソーシャルサーチサーチシステムでは，ページおよび閲覧者のランキング結果を検索エンジンに反映する．つまり，Nutch でクローリングしたリンク構造および，コミュニケーション機構より抽出した閲覧者情報を用いて，ランキングを行い，この結果を検索エンジン Solr に反映させる．この連携の実現にあたり以下の機能を実現する必要がある．

(a) クローリングにより抽出されたリンク構造を行列に変換する機能

(b) 求めたランク値を Solr に反映させる機能

(a) の行列変換機能は，Nutch が生成する linkdb を解析することで実現可能であると考えられる．取得した linkdb は各ページにおけるインリンクに対応する形で記されているため，3.1.2 節で述べた方法で遷移確率行列に変換できる．(b) では，Solr の検索の際，通常スコア計算は式 (5) で求めるが，4.2 で説明したが，式中の $norm(t, d)$ では，文書の重要度を含んでおり，この重要度に求めたランク値を反映することにより提案しているスコア値をつけることが可能であると考える．重要度はインデックス生成時に反映されており，検索エンジン Lucene のサブプロジェクトで開発されている Luke を用いることによりインデックス生成された情報を閲覧することができる．したがって，インデックス生成時に重要度をランク値として反映することにより提案するランク値が実現できると考える．

5 まとめと課題

本稿ではランキング機構におけるランク値計算手法の比較および，検索システムへの組み込みの検討を行った．ランク値計算は，固有値計算ライブラリ SLEPc と PageRank 計算をサポートしているライブラリ

JUNG を比較を行い，SLEPc の方が並列化されており，複数のプロセッサを用いて計算が可能であるため，性能が良い結果が得られた．また，JUNG で用いられていた計算手法のべき乗法は Google で使用されていた手法で，大規模データの分散処理を支える Hadoop の MapReduce を用いることにより，計算の高速化を実現できる可能性があり，今後調査する必要がある．

検索システムへの組み込み手法として，クローリングにより抽出されたリンク構造を行列に変換する機能，求めたランク値を Solr に反映させる機能を満たす必要があり，リンク構造については Nutch において取得が可能であるので，データ解析することにより，行列を生成できると考えられ，プログラミングする必要がある．また，Solr に反映させる機能においては，スコア計算を行われている Similarity クラスを理解することで，スコアリングの拡張を行い，求めたランク値を反映することが可能であると考え，調査する必要がある．

6 付録

6.1 PETSc および SLEPc のインストール方法と使い方

PETSc, SLEPc を実際に使用するために，評価環境 agile(表 1) へのインストールを行った．インストールには Ubuntu のパッケージ管理システムである apt-get を用いた．

apt-get を用いた PETSc, SLEPc のインストール

```
### PETSc のインストール
$ sudo apt-get install libpetsc3.0.0

### SLEPc のインストール
$ sudo apt-get install libslepc3.0.0
```

これで，PETSc, SLEPc のインストールが完了した．

インストールした PETSc, SLEPc を用いるには，コンパイル時に使用するライブラリなどを指定する必要がある．

例題として，`slepc-3.0.0-p7/src/examples/ex1.c` を例にとって説明する．ソースコードは http://www.grycap.upv.es/slepc/download/download.htm よりダウンロードすることができる．

まず，コンパイルの仕方であるが，コンパイル時にリンクする PETSc, SLEPc のライブラリを指定する必要がある．これにより，`example` という実行ファイルを得ることができる．

コンパイルの仕方

```
$ mpicc -I /usr/include/petsc -I/usr/include/slepc -lpetsc -lslepc \
  -o example ex1.c

### 簡単のため，エイリアスを設定しておくとよい
$ alias mpicc="mpicc -I /usr/include/petsc -I/usr/include/slepc
  -lpetsc -lslepc"
```

コンパイルによって得られた実行ファイル `example` を次のように実行する．

プログラムの実行方法

```
$ mpirun example
```

SLEPc では固有値計算手法として，Arnoldi, Krylov-Schur, Lancozs を利用することができるが，それらの指定は以下のようにソースコードに記述する必要がある．デフォルトで使用される計算手法は Krylov-Schur である．

固有値計算手法の指定

```
   --- 前略 ---
/*
   Set operators. In this case, it is a standard eigenvalue problem
*/
ierr = EPSSetOperators(eps,A,PETSC_NULL);CHKERRQ(ierr);
ierr = EPSSetProblemType(eps,EPS_HEP);CHKERRQ(ierr);

// 固有値計算手法の指定
ierr = EPSSetType(eps, EPSARNOLDI);CHKERRQ(ierr); // Arnoldi を使用
// ierr = EPSSetType(eps, EPSARNOLDI);CHKERRQ(ierr); // Lanczos を使用

/*
   Set solver parameters at runtime
*/
ierr = EPSSetFromOptions(eps);CHKERRQ(ierr);
   --- 後略 ---
```

同様に，許容誤差，最大繰り返し回数の指定は以下のようにソースコードに記述する．ここで，許容誤差は指数表現で，最大繰り返し回数は整数で指定する．

許容誤差，最大繰り返し回数の指定

```
   --- 前略 ---
/*
   Set operators. In this case, it is a standard eigenvalue problem
*/
ierr = EPSSetOperators(eps,A,PETSC_NULL);CHKERRQ(ierr);
ierr = EPSSetProblemType(eps,EPS_HEP);CHKERRQ(ierr);

// 許容誤差，最大繰り返し回数の指定
ierr = EPSSetTolerances(eps,1e-07,1000);CHKERRQ(ierr);

/*
   Set solver parameters at runtime
*/
ierr = EPSSetFromOptions(eps);CHKERRQ(ierr);
   --- 後略 ---
```

なお，許容誤差と最大繰り返し回数はオプション (**??**節) でも指定することができる．

6.2 mpirun で使用可能なオプション

mpirun で使用できるオプションには以下のようなものがある．

- -np <processors> : 使用するプロセッサ数の指定
  <processors> に使用するプロセッサ数を整数で指定する．
  (e.g. -np 4)
- -log_summary : ログの出力
  -log_summary オプションを用いると，プログラムの実行時間，事前設定した箇所の呼び出し回数や実行時間を得ることが出来る．
- -eps_max_it <iterations> : 最大繰り返し回数の指定
  <iterations> には最大繰り返し回数を整数で指定する．
  (e.g. -eps_max_it 1000)
- -eps_tol <tolerances> : 許容誤差の指定
  <tolerances> には許容誤差を指数表現で指定する．
  (e.g. -eps_tol 1e-07)
- -mat_view : 行列の要素を標準出力に表示
  -mat_view オプションを用いると，以下のような出力が得られる．
  row 1: (0, 0.5)

これは，1 行目 0 列の値が 0.5 であるということを表している．なお，記述されていないところはすべて 0 である．

- `-eps_monitor` : 収束具合を監視

6.3 JUNG の使い方

JUNG は Java 記述されているため，実験は実行からデバックまで可能である，Eclipse を用いて行った．使用方法は，JUNG のホームページ [7] より，ソースコードをダウンロードする．次に，Eclipse を立ち上げ，Eclipse のビルドパスに，ソースコードのパスを通すことで使用できる．以下は JUNG を用いて PageRank 計算を行うにあたって使用する主なクラスである．

JUNG のランク計算における主なクラス

```
# グラフの種類
DirectedSparseGraph    //有向グラフ
UndirectedSparseGraph //無向グラフ

# ノードの生成
graph.addVertex(ノード番号)

# エッジの生成
graph.Edge(new MyEdge(辺の重み，辺の名前)，始点，終点)

# PageRank
PageRank(graph，Edge 情報，ランダム値)
setMaxIteration(数値)  //最大繰り返し回数の設定
setTolerance(数値)     //許容誤差の設定
evaluate()             //計算開始
```

6.4 Nutch のインストールと使い方

Nutch のインストール方法を以下に記述する．

Nutch のインストールと設定

1. http://nutch.apache.org/より Nutch をダウンロードし${HOME}/local/に解凍する.
2. ant コマンドで build
3. .bashrc に以下を記述

```
export NUTCH_RUNTIME_HOME = ${HOME}/local/nutch-1.4/runtime/local/
export JAVA_HOME=/System/Library/Frameworks/JavaVM.framework/Versions/
1.6/Home/
```

4. $NUTCH_HOME/conf/nutch-site.xml に下記を追加

```
<property>
  <name>http.agent.name</name>
  <value>My Nutch Spider</value>
</property>
```

Nutch ではクローリングをすることができ，クローリング方法を下記に記述する.

クローリング方法

NUTCH のホームディレクトリに移動し，クローリングするページを urls.txt に記述

```
$cd $NUTCH_RANTIME_HOME
$mkdir url
$emacs url/urls.txt
 http://www.yahoo.co.jp
 http://www.google.co.jp
```

NUTCH のコマンドを用いてクローリング

```
$bin/nutch crawl urls -dir crawl -depth 5 -topN 10
```

また，クローリングの各過程をコマンドにり，一つずつ実行することができ，以下のコマンドを用いる.

クローリングのコマンド

```
# クローリングを一通り行う
$bin/nutch crawl urls -dir crawl -depth 5 -topN 10
# inject を行う
$bin/nutch inject crawl/crawldb urls
# generate を行う
$bin/nutch generate crawl/crawldb crawl/segments -topN 10
# fetch を行う
$s1="ls crawl/segments -1 | tail -1"
$bin/nutch fetch crawl/segmetnts/$s1
# updatedb を行う
$s1="ls crawl/segments -1 | tail -1"
$bin/nutch updatedb crawl/crawldb crawl/segments/$s1
# invertlinks を行う
$bin/nutch invertlinks crawl/linkdb -dir crawl/segmets
# index を行う
$bin/nutch index crawl/index crawl/crawldb crawl/linkdb crawl/segments/*
```

また，各過程で取得したデータの確認方法は以下である．

取得したデータの確認方法

```
#crawldb の dump
$bin/nutch readdb crawl/crawldb -dump dump
#segment の抽出結果の表示
$bin/nutch readseg -get crawl/segments/* 取得したい URL
#linkdb の dump
$bin/nutch readlinkdb crawl/linkdb -dump dump
```

6.5 Solr の設定

Solr の設定方法および起動方法は以下に記述する．

Solr の設定および起動方法

1. Solr を http://lucene.apache.org/solr/よりダウンロードし，${HOME}/local/に解凍をする．
2. .bashrc に以下を記述する
```
export APACHE_SOLR_HOME=${HOME}/local/solr-3.5.0/
```
3. 以下のコマンドを実行する
```
$cp $NUTCH_HOME/conf/schema.xml $APACHE_SOLR_HOME/example/solr/conf/
```
4. 以下のコマンドによりサーバを起動する
```
$cd $APACHE_SOLA_HOME/example/
$java -jar start.jar
```

Solr では，日本語の形態素解析は実装されていないため，日本語の解析を行う場合にはインストールをする必要がある．インストール方法は以下に記述する．

日本語の形態素解析の導入

1. 日本語の形態素解析する lucene-gosen を http://code.google.com/p/lucene-gosen/ よりダウンロードし，${APACHE_SOLR_HOME}/example/solr/lib に解凍する．
2. solr/conf/schema.xml に以下を記述する
```
<fieldType name="text_ja" class="solr.TextField"
                          positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.JapaneseTokenizerFactory"/>
  </analyzer>
</fieldType>
```

謝辞

本研究の進行，および，論文の執筆にあたり，1 年間半親身にご指導をいただいた秋山豊和准教授に感謝致します．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意見交流や，論文発表の際のご指摘ありがとうございました．最後に，同じ研究室の皆様には，困ったときに助けていただきありがとうございました．特に 4 回生の尾崎拓也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ませんでした．感謝しています．

参考文献

[1] 松井 優也，河合 由起子，秋山 豊和，青木 聡，” 検索とコミュニケーションによる知識獲得支援システムの検討”，IEICE Technical Report, Vol.110, No.430, IA2010.91, pp.57-62,(2011)

[2] goo ラボ　ぺちゃくちゃ検索,http://pechakucha.labs.goo.ne.jp/

[3] 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).

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

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

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

[7] JUNG: Java Universal Network/Graph Framework, http://jung.sourceforge.net/

[8] F. シャトラン 著，伊理 正夫，伊理 由美 訳，”行列の固有値 最新の解法と応用”, (2010)

[9] Jimmy Lin, Cbris Dyer 著, 神林 飛志, 野村 直之 監修, 玉川 竜司 訳, “Hadoop MapReduce デザインパターン MapReduce による大規模テキストデータ処理”, pp.103-124, (2011)

[10] 関口 宏司, 大谷 純, 三部 靖夫, 武田 光平, 中野 猛 著, “オープンソース全文検索エンジン Apache Solr 入門”

[11] Apache Solr: Welcome to Solr , http://lucene.apache.org/solr/