卒 業 論 文

大規模グラフ計算用言語 Fregel の
二部グラフマッチングへの応用と拡張

指導教員: 江本 健斗 准教授

九州工業大学情報工学部
知能情報工学科

2017 年度

栗木 駿一

所属部門	知能情報アーキテクチャ	指導教員	江本 健斗　准教授
学生番号	14231025	氏　名	栗木　駿一
論文題目	大規模グラフ計算用言語 Fregel の 二部グラフマッチングへの応用と拡張		

1　はじめに

近年，SNS におけるユーザー間の関係や，Web ページ間のリンクなどの，大規模なグラフの解析を行う機会が増加してきている．これを受けて，大規模グラフ計算用言語 Fregel が提案された [1]．Fregel は頂点ごとに独立して計算を実行する頂点主体モデルを採用しており，頂点が実行する処理のみを記述することでグラフ計算を並列に行うことができる．しかし，Fregel においての実際のグラフ計算問題の記述例は少ない．

本研究は，Fregel の記述例を増やしてその有用性を検証するために，二部グラフマッチングの Fregel での記述を行った．また，現状の Fregel では機能が不足していたため，Fregel の拡張も同時に行った．

2　二部グラフマッチングの頂点主体計算

二部グラフマッチングとは，頂点が左右の二つの集合に分けられ，それぞれの集合内の頂点間には辺が存在しないようなグラフ（二部グラフ）において，各頂点が高々一つの（他方の）頂点と繋がるような辺の部分集合，ないし，それを求める計算のことである．

これを頂点主体で行うには，まず，左集合の頂点がランダムに右集合の頂点を一つ選択する．次に，右集合の頂点が，左集合の頂点から選ばれたかを調べ，選ばれていればその頂点を選んだ左集合の頂点の中から一つをランダムに選択する．最後に，左集合の頂点が右集合の頂点から選ばれているか確認し，選ばれていればその頂点とマッチングする．これを全頂点がマッチングするか，選択できる頂点が無くなるまで繰り返す．

3　Fregel による記述と拡張の提案

本研究で，実際に二部グラフマッチングを Fregel で記述したものが図 1 である．入力グラフに ss0，ss1，ss2 の 3 つの操作を順に繰り返し行っていくことで，二部グラフマッチングの計算を実現している．

この計算の記述には，リストからランダムに要素を取り出す，といった集約関数が必要となる．しかし，現状の Fregel ではその機能が不足していたため，本研究で新しい集約関数 random を導入するという拡張を行った．これは，random x list と記述することで，list が空ならば x を返し，list に要素があるならばその中からランダムに一つ取り出す，といった関数である．

```
1  bipartitematching g =
2   let init v = (-1);
3       ss0 v = if(vid v > size/2 && val v.^set == -1)
4               then random (-1) [vid u|(e,u)<-is v,
5                                 val u.^set == -1]
6               else val v.^set);
7       ss1 v = if(vid v <= size/2 && val v.^set == -1)
8               then random (-1) [vid u|(e,u)<-is v,
9                                 val u.^set == vid v]
10              else val v.^set);
11      ss2 v = if(vid v > size/2 && val v.^set /= -1)
12              then random (-1) [vid u|(e,u)<-is v,
13                                val u.^set == vid v]
14              else val v.^set);
15      step g = let g1 = gmap ss0 g;
16                   g2 = gmap ss1 g1;
17                   g3 = gmap ss2 g2
18               in g3
19    in giter init step Fix g
```

図 1 :二部グラフマッチングの Fregel コード

4　実験と評価

図 1 のコードに対し，その並列実行を確認する実験を行った．入力にはランダム生成した，各頂点が 10 本の辺を持つ二部グラフを使用した．頂点数毎に，計算機の台数に応じた速度向上の様子を図 2 に示す．

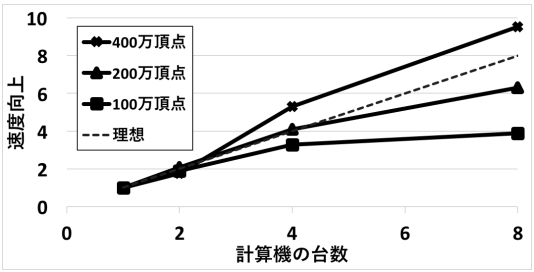

図 2 :並列実行による速度向上の様子

図 2 から，計算機の台数が増加するに連れて速度が向上していることがわかる．また，頂点数が増加するに連れて速度向上が改善しているため，より大規模なグラフほど並列化の効果が顕著になることが分かる．よって，本研究で拡張した機能を用いたグラフ計算においても，十分な並列性能が得られることが分かった．さらに，少しの機能拡張により二部グラフマッチングを記述できたことで，Fregel の記述性も確認された．

参考文献

[1] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing. In *Proceedings of ICFP 2016*, pages 200–213, 2016.

目 次

第1章　はじめに

近年，SNS におけるユーザー間の関係や，Web ページ間のリンクなどの，大規模なグラフの解析を行う機会が増加してきている．例えば，SNS のユーザーを頂点，ユーザー間のフォロー/フォロワー関係を辺とすると，SNS は大規模なグラフと考えることができる．このグラフに対して解析を行うことで，そのユーザーに関係のあるユーザーを提示したり，ユーザーが興味を示す情報を提示することができる．このような大規模グラフを扱う計算を実行しようとすると，単一の計算機では時間がかかるため，複数の計算機を用いて並列処理を行う必要がある．

そこで，大規模グラフを並列処理するためのフレームワークとして Pregel [1] が提案されている．Pregel は頂点主体の計算モデルを採用しており，ユーザーは各頂点で繰り返される単一の計算を記述するだけで，グラフ計算を並列に行うことができる．しかし，Pregel の問題点として，複雑なアルゴリズムを記述する場合にはプログラミングが難しい，ということが挙げられている．この問題点を解決するために Pregel モデルを高階関数として定式化し，関数型の領域特化言語として実現した Fregel [2] が提案された．Fregel で記述されたプログラムは Pregel に変換することができ，現在は Pregel のオープンソース実装である Apache Giraph へのコンパイラが用意されている．しかし，この Fregel でのグラフ計算問題の記述例はまだまだ数が少ない．

本研究は，Fregel の記述例を増やし，その有用性を検証するために，二部グラフマッチングの Fregel での記述を行った．また，現状の Fregel ではこのグラフ計算を実現するための機能が不足していたため，Fregel の機能の拡張を行った．拡張した機能を用いて二部グラフマッチングを記述し，並列実行により実験を行なった結果，十分な並列性能を確認することができた．

以降の論文の構成について述べる．２章では，本研究で利用する Pregel や Fregel について説明する．３章では，二部グラフマッチングについてと，Pregel による二部グラフマッチングのアルゴリズムについて述べる．４章では，Fregel による二部グラフマッチングの記述と，それに必要となった Fregel の拡張の提案を行う．５章では，実際に大規模なグラフで二部グラフマッチングを並列実行した実験の結果を述べる．６章では，本研究についてまとめる．

第2章　準備

2.1　Pregel

Pregel [1] は Google が提唱した大規模グラフを並列処理するためのフレームワークである．Bulk Synchronous Parallel（BSP）に基づく頂点主体の計算モデルを提供している．このモデルでは，ユーザーは各頂点の振る舞い（頂点計算）を記述するだけでグラフ計算のための並列プログラムを作ることができる．

2.1.1　概要

Pregel では，各頂点がユーザーの記述した同一の処理を実行すること (superstep) と，全ての頂点での処理の終了を待つこと（バリア同期）の繰り返しによってグラフ計算が実行される．この繰り返しは，グラフ計算が収束するまで実行される．また，頂点での処理実行は全頂点で並列に実行される．

2.1.2　メッセージの送受信

頂点間の通信はメッセージの送受信によって行われる．superstep S の処理を実行する時，頂点は一つ前の superstep S-1 の処理にて送信されたメッセージを参照することができる．また superstep S の処理で送信されたメッセージは次の superstep S+1 で利用することができる．このように頂点が送信したメッセージは次の superstep でのみアクセスできる，といった制限を設けることで通信によるデッドロックを回避することができる．これはある頂点が必要とするメッセージはバリア同期によって送信されていることが保証されているからである．メッセージの送信は頂点 ID に基づいて行われるが，基本的にメッセージの送信はその頂点の隣接点への送信である．

2.1.3 計算の終了判定

Pregel の計算は superstep とバリア同期の繰り返しで，全ての頂点での計算が収束（終了）するまで繰り返される．その収束を判定するため，各頂点は Active と Inactive という二つの状態を持つ．全ての頂点が Inactive になることでグラフ全体の計算を終了する．Active な頂点はユーザーの記述した処理を実行し，Inactive な頂点は実行しない．処理の中で voteToHalt と呼ばれる操作が行われると，その頂点の状態が Active から Inactive へ変化する．また Inactive な頂点はメッセージを受信することで Active 状態となる．

2.1.4 Pregel での計算例

Pregel での計算の例として，ある頂点から到達可能な頂点を調べる到達可能性問題のコードを Listing 2.1 に示す．このプログラムは有向グラフを入力とする．頂点と受信メッセージを入力として受け取り，始点から到達可能（true）とマークされた頂点の隣接頂点を true とマークしていく．このコードは主に初期化部分 (2-5 行目) と繰り返しの部分 (6-10 行目) に分けることができる．

最初の superstep では，計算を実行している頂点（v）が始点かどうか判別し，始点ならば true とマークしその情報を隣接頂点へ送信する．以降の superstep では，受信したメッセージに true が含まれていて，かつ，まだ自身が false であるならば，自身を true とマークしてそのことを隣接頂点へ伝える．最後に voteTohalt によって頂点を Inactive にする．

```
1  vertex.compute(v,message){
2    if(superstep == 0){
3      v.rch = v.vid == 0;
4      if(v.rch)
5        sendToNeighbors(v.rch);
6    }else{
7      newrch = v.rch || or(message);
8      if(newrch != v.rch){
9        v.rch = newrch;
10       sendToNeighbors(newrch);
11     }
12   }
13   voteToHalt();
14 }
```

Listing 2.1: Pregel のプログラム例

図 2.1 に Pregel モデルを用いた到達可能性問題を解く過程のグラフを示す．左が入力グラフであり，残りの右のグラフは表示されている回数の superstep 終了後

のグラフを表している．白の頂点は Active な状態，灰色の頂点は Inactive な状態であることを示す．

初めに，superstep0 では始点である頂点に true がマークされ，隣接している 2 つの頂点へ true（T）のメッセージが送信される．superstep1 ではメッセージを受信した 2 つの頂点は active 状態となり，始点は Inactive となる．ture のメッセージを受信した頂点は ture をマークし，隣接している頂点へ ture を送信する．superstep2 で最後の頂点に true がマークされる．superstep3 で全ての頂点が Inactive の状態になるため計算が終了する．

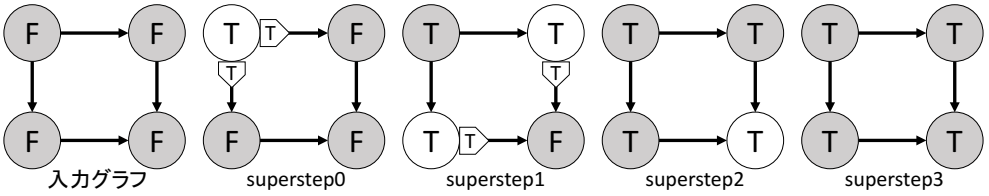

図 2.1: Pregel の計算過程の様子

2.1.5 Pregel の問題点

Pregel プログラミングは採用している BSP モデルの通信に関する制約から，任意のグラフ計算を必ずしも簡単に書けるものではない．これには，主に 3 点の理由が挙げられる．

まず 1 つ目に，明示的な状態制御の難しさである．隣接した頂点や全頂点，グラフ全体の情報を集約し，その情報を用いて計算を行いたい場合がある．その場合，異なる superstep で異なる振る舞いをするように，superstep の回数で判定するなどして状態の制御を明示的に行われなければならない．

２つ目に, 明示的なメッセージの送受信の問題がある．頂点間の通信はメッセージの送受信によって行われるが，その superstep で使いたい情報は 1 つ前の superstep で送信しなければならない．複雑なプログラムになってくるとメッセージを送信する箇所，受信したメッセージを使用する箇所などを明確に把握していく必要がある．そのため，複雑なプログラムになるにつれてメッセージのやり取りや制御の流れの把握が困難になる問題点が挙げられる．

最後に，３つ目の問題点として明示的な計算停止制御が挙げられる．Pregel では計算を終了させるために各頂点の処理として，voteToHalt を呼び出す必要がある．したがって，全頂点の計算を適切に終了させるためには各頂点における処理を正しく記述する必要がある．しかし，複雑なプログラムになるにつれて正しく

処理を追うことが困難になってくる．そのため，予期せぬところで計算が停止したり，voteToHalt 操作が正しく呼び出されず計算が止まらないなど，適切な場面で計算を終了させることが困難になってくる．

2.2 Fregel

Fregel [2] とは，Pregel の問題点を解決しようと提案された，大規模グラフ処理のための関数型領域特化言語である．

2.2.1 概要

Fregel は Pregel モデルを基本に設計されている．Haskell プログラムとして動作可能であり，Haskell インタプリタを用いた動作確認やデバックが可能となっている．また，高階関数を用いたグラフ計算の記述が可能となっており，この高階関数はひとつの Fregel プログラムの中で複数回使用することができる．Pregel では頂点間の通信の表現として poke-base の通信の表現を採用していたが，Fregel では peek-base の通信の表現を採用している．これにより，頂点間での必要な情報のやり取りを，より直感的に記述することができるようになっている．

2.2.2 Fregel における通信の表現

Fregel では，peek-base の通信の表現を用いることで，Pregel の問題点として挙げた明示的なメッセージの送受信の問題を解決している．Pregel における poke-base，Fregel における peek-base の通信の表現を図 2.2 に示す．頂点 v での頂点計算を行う場合に，隣接している頂点 u の情報を使用したい場合を考える．Pregel における poke-base の通信では使用したい頂点 u の情報を頂点 v へ送信するように記述する必要がある（例：Listing 2.1 の 10 行目）．しかし，Fregel における peek-base の通信では，頂点 v が使いたい頂点 u の情報を自分から見に行く，といった振る舞いになるため明示的な通信の記述をする必要がない（2.2.5 節で例示する）．

2.2.3 グラフ高階関数

Fregel は様々なグラフ処理を簡単に記述するために次の 4 つの高階関数を提供している．関数 fregel は Pregel の関数的モデルを実装し，初期化関数，ステップ関数，終了条件を受け取る．関数 gzip は 2 つのグラフを受け取り，対応する頂点

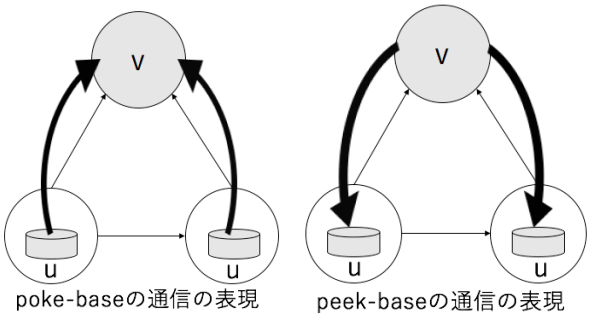

図 2.2: 通信の表現

をペアとする．関数 gmap はグラフの全頂点に与えられた関数を適用する．関数 giter は関数 fregel と同様に初期化関数，ステップ関数，終了条件を受け取る．この関数はグラフを受け取り，そのグラフに処理を適用した後のグラフを返すものである．そして受け取った終了条件が満たされるまで繰り返し関数の適用を繰り返す．

2.2.4 終了条件

高階関数 fregel，giter が受け取る終了条件として Fix，Until，Iter の 3 つが用意されている．Fix はグラフの各頂点の値が変化しなくなる不動点に達するまで計算を行う．Until は式を引数として受け取り，その式が満たされるまで計算を実行する．Iter は整数を引数として受け取り，その回数分計算を実行する．

2.2.5 Fregel での計算例

例として Listing 2.2 に到達可能性問題の Fregel プログラムを示す． 1 行目では頂点が持つデータ型を定義している．そして， 2 行目以降の関数 reAll が主要な部分となる．init 部分が初期化関数にあたり，その頂点が始点である場合のみ rch フィールドの値が True である Rval レコードを返す．頂点の ID は特殊なプリミティブ関数 vid で取得している．ステップ関数 step は直前の rch フィールドに格納されている結果を隣接頂点から集約している．これは is v を生成子とする内包表記で実現している．この結果と自分の値の論理和を or 関数で求めることで，隣接している頂点が到達可能ならば自身も到達可能であるとマークすることができる． 7 行目は高階関数 fregel の引数が上で定義した初期化関数 init，ステップ関数 step，終了条件の Fix，グラフ g となっている．

```
1 data Rval = Rval { rch :: Bool } deriving Eq
2  reAll g =
3    let init v = Rval(vid v == 0)
4        step v prev curr =
5         let newrch = prev v.^rch||or [prev u.^rch | (e,u) <- is v]
6        in Rval newrch
7    in fregel init step Fix g
```

Listing 2.2: Fregel のプログラム例

2.3 Apache Giraph

Apache Giraph [3] とは，オープンソースのフレームワークで，Apache Hadoop [4] と呼ばれる分散処理環境で実行されるシステムである．Java 言語により実装されており，Giraph プログラムも Java 言語を使って記述する．本研究の実験では Fregel で作成したコードを Giraph にコンパイルし，動作させる．

第3章　二部グラフマッチングの頂点主体計算

3.1　二部グラフマッチング

二部グラフとは，図 3.1 のような，頂点の集合が左右の二つの部分集合に分けられ，それぞれの集合内での頂点間には辺が存在しないようなグラフのことである．グラフのマッチングとは，各頂点が高々一つの頂点と繋がるような辺の部分集合，ないし，それを求める計算のことである．

二部グラフマッチングは二部グラフにマッチングを適用したもので，図 3.2 の太くなっている辺が例として挙げられる．マッチングは一つのグラフにおいて何通りか存在する場合もある．本研究で扱うアルゴリズム [1] は，必ずしも最大数のマッチングを得ることができないが，ある程度の大きさのマッチング数を確保するものとなっている．

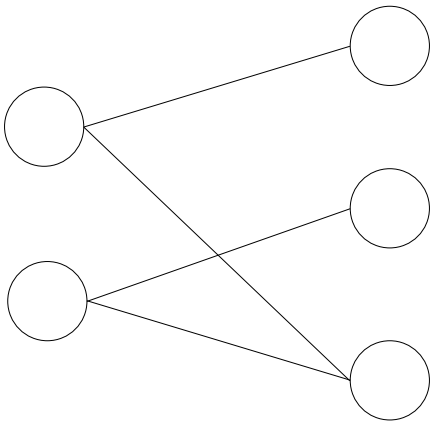

図 3.1: 二部グラフの例

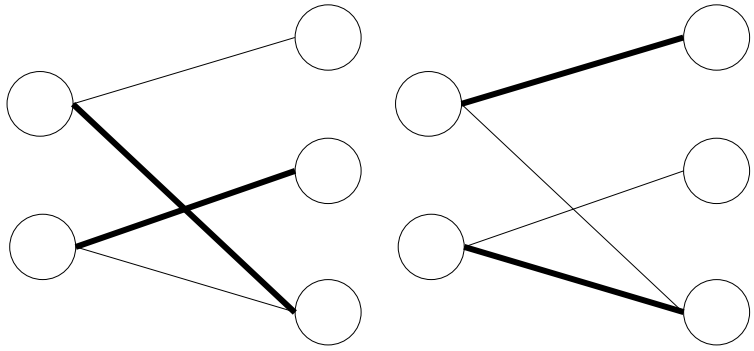

図 3.2: 二部グラフマッチングの例

3.2 Pregel アルゴリズム

二部グラフマッチングを Pregel で記述したものが Listing3.1 である．このアルゴリズムは 4 回の superstep を 1 サイクルと考え，全頂点がマッチングするかマッチングできる頂点が無くなるまでサイクルを繰り返す．各頂点は，自身のマッチング先の頂点 ID を保持するものとなっている．

superstep0 で，自身が保持しているマッチング先の ID の初期化として-1 を入れる（4-6 行目）．サイクル内の 0 番目での superstep（case0）では，片方の集合の各頂点が，隣接している頂点全てにメッセージとして自分の頂点 ID を送信する (6-12 行目)．次の superstep（case1）では，他方の集合の各頂点が受け取ったメッセージからランダムに一つ選び，選んだ頂点 ID にメッセージとして自分の頂点 ID を，選ばななかった頂点 ID に-1 を送信している (13-40 行目)．そして，その次の superstep（case2）では，まず，受け取ったメッセージの正負を確認する．正であればメッセージの値を自分の値に代入し，その頂点 ID へメッセージとして自分の頂点 ID を送信する．負であれば，選ばれなかったということなので値は-1 のままにし，次のサイクルで再度マッチングを探す．もし正の値であるメッセージが複数存在した場合は，その中からランダムに一つ選ぶようにする (41-62 行目)．サイクル内最後の superstep（case3）で，受け取ったメッセージを自分の値に代入して 1 サイクルが終了となる (63-73 行目)．次のサイクルが終わっても値が前のサイクルの値のままであれば，それ以上マッチングは存在しないということなので全体の計算が終了となる．

```
public void compute(
  if(getSuperstep() == 0){
    vertex.setValue(new DoubleWritable(-1));
  }
  switch((int)getSuperstep() % 4) {
  case 0:
    if(vertex.getId().get()%2 == 0 && vertex.getValue().get() < 0) {
      double id = vertex.getId().get();
      sendMessageToAllEdges(vertex,new DoubleWritable(id));
      vertex.voteToHalt();
    }
    break;
  case 1:
    if(vertex.getId().get()%2 == 1 && vertex.getValue().get() < 0) {
      int mscount = 0,i = 0;
      double ms = 0;
      double id = vertex.getId().get();

      for(DoubleWritable message : messages) {
        mscount++;
      }
      if(mscount != 0){
        Random rnd = new Random();
        int rand = rnd.nextInt(mscount);

        for(DoubleWritable message : messages) {
          if(i == rand) {
            ms = message.get();
            sendMessage(new LongWritable((long)ms),new DoubleWritable(id));
          }
          else{
            ms = message.get();
            sendMessage(new LongWritable((long)ms),new DoubleWritable(-1));
          }
            i++;
          }
      }
    }
    vertex.voteToHalt();
    break;
  case 2:
    if(vertex.getId().get()%2 == 0 && vertex.getValue().get() < 0) {
      int mscount = 0,i = 0;
      long ms = 0;
      double id = vertex.getId().get();
      ArrayList<Long> mstrue = new ArrayList<Long>();
      for(DoubleWritable message : messages) {
        ms = (long)message.get();
        if(ms >= 0){
          mstrue.add(ms);
          mscount++;
        }
      }
      if(mscount != 0) {
        Random rnd = new Random();
        int rand = rnd.nextInt(mscount);
        sendMessage(new LongWritable(mstrue.get(rand)),new DoubleWritable(id));
        vertex.setValue(new DoubleWritable((double)mstrue.get(rand)));
        vertex.voteToHalt();
      }
    }
```

```
62      break;
63    case 3:
64      if(vertex.getId().get()%2 == 1 && vertex.getValue().get() < 0) {
65        double ms = vertex.getValue().get();
66        for(DoubleWritable message : messages) {
67          ms = message.get();
68        }
69        vertex.setValue(new DoubleWritable(ms));
70        vertex.voteToHalt();
71      }
72      break;
73  }
```

Listing 3.1: Pregel の二部グラフマッチング

Listing3.1 の Pregel の記述例では，頂点の実行する処理が case 毎に分割されている．メッセージの送信部分や送信されたメッセージを使用している部分が case を跨いでいるため，どのメッセージを送信しているのか，どのメッセージを使用したいのか正確に把握しながら記述していく必要がある．こういった点が第 2 章での問題点の例として挙げられる．

第4章　二部グラフマッチングの Fregelによる記述と拡張

4.1　Fregelアルゴリズム

本研究で，実際に二部グラフマッチングを Fregel で記述したものが Listing4.1 である．入力グラフの各頂点に対して初期化関数の init を適用した後，ss0，ss1，ss2 の処理を各頂点で繰り返し行うことで二部グラフマッチングを実現している．

Pregel では，頂点が受信したメッセージをランダムに選択する，という考え方であった．これに対し，Fregel では，周りの頂点の値を参照し，それらの集約した結果をランダムに選択する，といった考え方となる．

```
1  data SVal = SVal {set :: Int} deriving (Eq, Show)
2  bipartitematching g =
3   let init v = SVal(-1);
4       ss0 v = SVal(if (vid v'mod'2 == 0  && val v.^set == -1)
5                    then random (-1) [vid u|(e,u)<-is v,val u.^set == -1]
6                    else val v.^match);
7       ss1 v = SVal(if (vid v'mod'2 == 1 && val v.^set == -1)
8                    then random (-1) [vid u|(e,u)<-is v,val u.^set == vid v]
9                    else val v.^match);
10      ss2 v = SVal(if (vid v'mod'2 == 0 && val v.^set /= -1)
11                   then random (-1) [vid u|(e,u)<-is v,val u.^set == vid v]
12                   else val v.^match);
13      step g = let g1 = gmap ss0 g;
14                   g2 = gmap ss1 g1;
15                   g3 = gmap ss2 g2
16               in g3
17   in giter init step Fix g
```

Listing 4.1: Fregel の二部グラフマッチング

まず ss0 では片方の集合に存在する頂点が，繋がっている頂点全ての頂点 ID を集約し，その中からランダムで 1 つを選択する (4-6 行目)．次に，ss1 で，自分の頂点 ID を持っている頂点の ID を集約し，複数存在するならばその中からランダムに 1 つ選択する (7-9 行目)．最後に，ss2 で自分の頂点 ID を持っている頂点が存在するか調べ，存在していればその頂点とマッチングとなる (10-12 行目)．なお，以上のランダム選択において，選択対象が存在した場合には-1 が選ばれる．

関数 gmap は各処理をグラフに適用し，適用後のグラフを出力するものである．例えば 13 行目では，入力グラフ g に ss0 で定義した処理を行い，グラフ g1 として出力している．

4.2 Fregel の拡張

実際に記述した二部グラフマッチングだが，Listing 4.1 の 5 行目などにある集約関数 random は，既存の Fregel には機能として存在しなかった．そこで，本研究で新しい集約関数 random を導入するという拡張を行った．この集約関数は ramdom x list と記述することで，list が空ならば x を返し，list に要素があるならばその中からランダムに一つ取り出す，といった関数である．

新たに加えた集約関数 random のコンパイルについて述べる．コンパイル後のコードで使用する演算子のクラスの実装を Listing4.2 に，その機能を使用した部分を Listing4.3 に示す．

```
1  static class Choicer{
2    int n;
3    Choicer(){
4      n = 0;
5    }
6    int choice(int x, int y) {
7      n++;
8      return Math.random() <= 1.0/n ? y : x;
9    }
10 }
```

Listing 4.2: ランダム選択のオペレーターの準備

```
1  Choicer choicer = new Choicer();
2    for (MsgData msg : messages)
3      agg_X431 = choicer.choice(agg_X431, (msg.agg_X431).get());
```

Listing 4.3: ランダム選択を使用した箇所

集約関数 random を使用する際に choicer というオペレーターを用意することとした．これは，引数に x，y を取り，y が 1/n の確率で選ばれるものである．この整数 n は，渡された y の数をカウントしており，最初の y は 1/1 の確率で，次の y は 1/2，さらに次は 1/3 となっていくため，どの要素も最終的には同じ確率で選ばれるものとなっている．

この拡張を用いたグラフ計算でも Fregel の有用性があるかを検証するため，Listing3.1，Listing4.1 のプログラムを用いて実験を行なった．

第5章　実験と評価

5.1　実験環境

実験環境として，表5.1に示すサーバー8台からなるクラスタを使用した．サーバー同士はギガビットイーサネットで接続している．

表 5.1: 実験環境

OS	Ubuntu 14.04.5 LTS
CPU	Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
メモリ	16GB (8GB x2, PC4-17000)
JAVA	Oracle JDK 1.8.0 131
Hadoop	Version 1.2.1
Giraph	Version 1.1.0

入力グラフには，ランダムに生成した，各頂点が10本の辺を持つ二部グラフを使用した．100万頂点，200万頂点，400万頂点の三種類のグラフを用意し，それぞれで計算機の台数を変化させて実験を行なった．

5.2　二部グラフマッチングの並列実行の比較

Pregel，Fregelそれぞれで入力グラフの頂点数と計算機の台数を変化させた時の実行時間を表5.2，表5.3に示す．また，計算機1台の速度を基準とし，台数を増やした際の速度向上の様子を図5.1，図5.2に示す．

計算機台数	1	2	4	8
100 万頂点	29.9	25.2	16.4	15.4
200 万頂点	50.9	52.6	22.4	17.1
400 万頂点	93.0	167.7	36.3	23.4

表 5.2: Pregel コードの実行時間 (s)

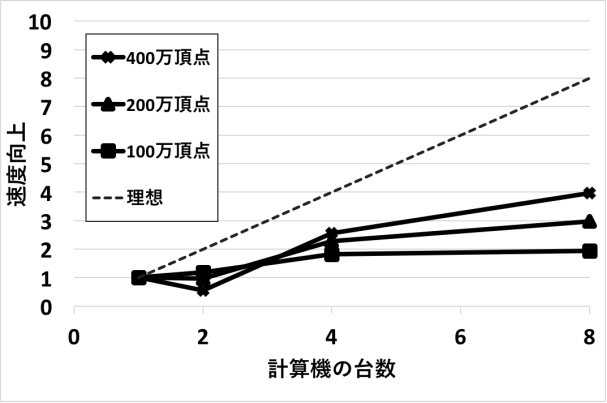

図 5.1: Pregel コードの速度向上の様子

計算機台数	1	2	4	8
100 万頂点	101.5	121.7	39.2	26.1
200 万頂点	210.0	404.3	75.6	39.1
400 万頂点	521.7	1555.5	200.8	71.4

表 5.3: Fregel コードの実行時間 (s)

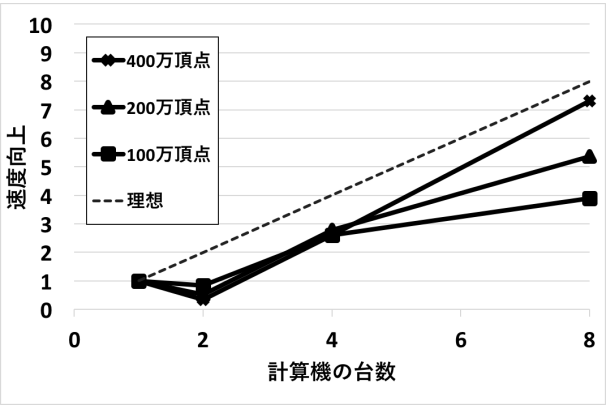

図 5.2: Fregel コードの速度向上の様子

図 5.1，図 5.2 から，計算機の台数が 4 台以上の時には理想値には届いてないものの，速度が向上していることがわかる．しかし，計算機の台数が 2 台の時は頂点数が増加するにつれて，速度が低下していることが確認できる．理由として，二部グラフの頂点集合の分け方が原因であると考えられる．

本実験では，頂点 ID の奇数，偶数により集合を区別している．このプログラムにおいて，頂点間の通信は異なる集合に属する頂点間のみで行われる．しかし，計算機 2 台で並列実行した際，片方の計算機に頂点 ID が奇数の頂点集合，もう片方の計算機に偶数の頂点集合が割り当てられ，計算機同士の通信量が増加したために速度が出なかったと思われる．これを確かめるため，追加実験を行うこととした．

集合の区別方法を変更し，再度実験を行なった．表 5.4 と表 5.5 に実行時間，図 5.3 と図 5.4 に速度向上の様子を示す．この実験では，頂点 ID が全頂点数の半分以下であるかどうかで頂点集合を区別している．

計算機台数	1	2	4	8
100 万頂点	29.0	20.5	15.8	15.4
200 万頂点	50.0	29.2	20.8	17.0
400 万頂点	91.0	49.8	31.5	22.5

表 5.4: Pregel コードの実行時間 (s)

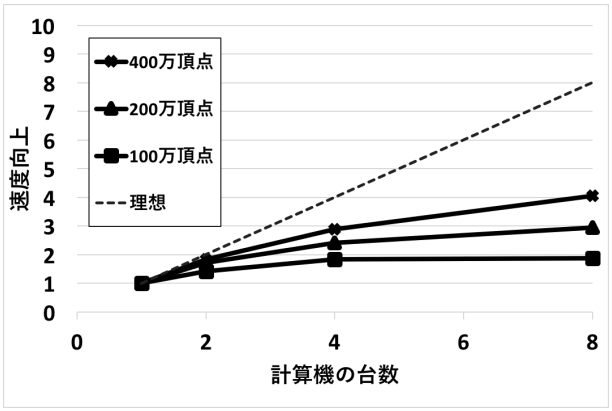

図 5.3: Pregel コードの速度向上の様子

計算機台数	1	2	4	8
100 万頂点	100.1	53.3	30.6	25.8
200 万頂点	201.2	97.5	49.0	32.0
400 万頂点	490.2	280.4	92.2	51.4

表 5.5: Fregel コードの実行時間 (s)

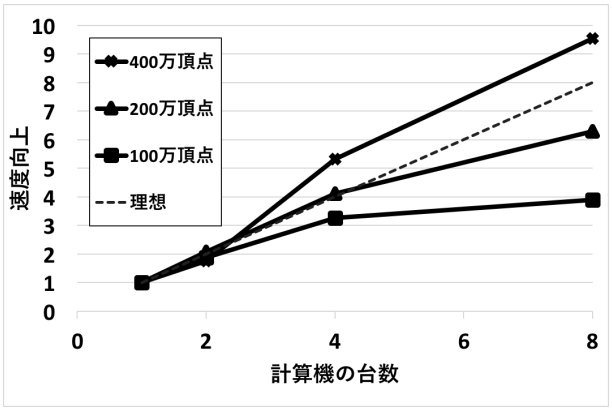

図 5.4: Fregel コードの速度向上の様子

図 5.3，図 5.4 から，どの計算機の台数においても台数が増加するにつれて速度が向上していることがわかる．また，頂点数が増加するにつれて速度向上が改善することが確認できる．Fregel コードを 400 万頂点で実行した際には速度向上が理想以上に改善されているが，これはもはや 1 台で処理するにはグラフが大きすぎるからであり，大規模なグラフほど並列化の効果が顕著になることがわかる．よって，本研究で拡張した機能を用いたグラフ計算においても，十分な並列性能が得られることがわかった．

第6章　まとめ

本研究では，二部グラフマッチングを題材に，Fregel の機能の有用性の検証と機能拡張を行なった．結果として，Fregel の拡張した機能を用いたグラフ計算においても十分な並列性能が確認できた．また，集約関数 random の拡張により二部グラフマッチングを記述できたことで，少しの機能拡張によりグラフ計算を記述できるという Fregel の記述性も確認された．

今後の課題として，次のようなことが考えられる．まず，他のグラフ計算を記述する際に必要な機能がまだ不足しているため，Fregel の機能のさらなる拡張が必要となるだろう．さらに，実際の実行時間を見てみると Fregel は Pregel と比べて十分に早いとは言えない．Fregel におけるグラフ計算の速度向上も課題の一つである．

謝辞

本研究を進めるにあたり，指導教員の江本健斗准教授には様々なご指導を賜りました．また，研究室の先輩，同級生には様々な助言を頂きました．関わってくださった皆様に感謝の意を示します．

参考文献

[1] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 135-146, 2010.

[2] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing. In Proceedings of ICFP 2016, pages 200-213, 2016.

[3] Apache Giraph. https://giraph.apache.org/

[4] Apache Hadoop. https://hadoop.apache.org/

[5] 惠羅博，土屋守正，『増補改訂版 グラフ理論』，産業図書，2011.