大規模グラフ並列処理のための関数型領域特化言語 Fregel とその評価

江本 健斗　松崎 公紀　胡 振江　森畑 明昌　岩崎 英哉

近年，大規模グラフ処理を実現する並列プログラムの開発が重要となっている．頂点主体のグラフ並列計算モデルである Pregel が Google によって提唱され，それに基づくプログラミング環境も提案されている．しかし，Pregel モデルでは，頂点間の直接通信の使用やその通信毎の頂点計算の分割記述などの低レベルなプログラミングが必要とされ，また，全体の計算をひとつの関数として記述しなければならないため，複雑な計算を行うプログラムは非常に煩雑になってしまう．この問題に対し，我々は，Pregel モデルの計算を高階関数の組み合わせによって記述することを可能とする関数型領域特化言語 Fregel を提案する．Fregel は，直接参照的な記述による頂点間データアクセスとグラフ計算高階関数群による宣言的でコンポジショナルなプログラミングを特徴とする．Fregel コンパイラは，与えられたプログラムから通信や同期を自動的に抽出する．本発表では，Fregel の基本設計と特徴，実問題に対するプログラミング例，試作コンパイラを用いた実験結果について報告する．なお本発表は ICFP2016 で発表予定の成果を拡張したものである．

1 はじめに

近年，日常的な様々な場面で，大規模なグラフが身近な存在になってきている．それに伴い，グラフを対象とする効率的な並列処理を可能とするソフトウェア，およびそのためのプログラミングモデルの必要性が増している．中でも Google による Pregel [7] は，グラフ分散処理のためのスケーラブルなモデルとして注目されており，GPS [8]，Giraph [1]，Pregel+ [12] などのオープンソースの実装も作られている．Pregel の特徴は，バルク同期並列 (Bulk Synchronous Parallel, 以下 BSP) [11] に基づく頂点主体の計算モデルを採用している点である．このモデルでは，処理対象のグラフの頂点がクラスタ中のノードに分散して配置され，各頂点は互いにメッセージを送受信しながら superstep と呼ばれる計算 (以下 SS と呼ぶ) を独立かつ同期的に繰返して処理を進める．

簡単な例として，与えられたグラフに対して，出発頂点 (頂点番号 0) から到達可能なすべての頂点を求める問題を考える．以後，この問題を「全到達可能性問題」を呼ぶ．頂点主体の計算でこの問題を解くには，全頂点をはじめに `false` とマークした上で，図 1 に示すプログラムを SS として全頂点で繰返し実行する．このプログラムは，頂点と受信メッセージを入力として受け取り，出発頂点から到達可能 (`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．最初の SS (2 行目) では，出発頂点だけを `true` とマークし，この情報を隣接頂点に送信する．以降の SS (6 行目) では，各頂点は `true` を含むメッセージが送信されているか，および，自身がまだ `false` であるかを確認し，もしそうであるならば自身を `true` とマークして (9 行目)，そのことを隣接頂点に知らせる (10 行目)．

Pregel コードの実行過程において，各頂点は実行中 (active) か停止中 (inactive) のいずれかであり，初期状態では，すべての頂点は実行中である．実行中の頂

Concurrent Operations on Splay Trees.
Kento Emoto, 九州工業大学, Kyushu Institute of Technology.
Kiminori Matsuzaki, 高知工科大学, Kochi University of Technology.
Zhenjiang Hu, 国立情報学研究所, National Institute of Informatics.
Akimasa Morihata, 東京大学, The University of Tokyo.
Hideya Iwasaki, 電気通信大学, The University of Electro-Communications.

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

図 1　全到達可能性問題に対する Pregel 風のコード.

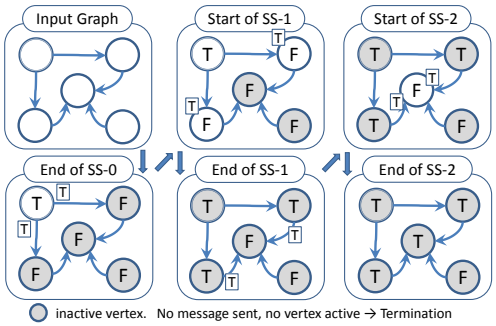


図 2　全到達可能性問題の具体例.

点は `voteToHalt` 関数 (5 行目，12 行目) を呼ぶことにより停止中に移行する．各 SS では，実行中の頂点だけが計算に参加し，必要に応じてメッセージを他頂点に送信する．停止中の頂点は，他頂点からメッセージを送信されると再び実行中になる．全頂点が停止中になり，未受信のメッセージが残っていなければ，グラフ全体の計算が終了する．図 2 に，5 頂点のグラフに対して，図 1 のプログラムによって全到達可能性問題を解く過程を示す．この過程では，3 回の SS で計算が終了する．

頂点主体の計算によるグラフ処理は強力ではあるが，手続き的な記述を行い，さらに明示的なメッセージ送受信を行わなければならないので，プログラムが非直観的でわかりにくくなる傾向がある．実際，図 1 のプログラムにおいて，次のような問題点を見ることができる．

- **副作用に依存したコード**　`vertex.compute` は，頂点に局所的な値を更新したり暗黙的な送信路を通してデータを明示的に送信している点において，副作用に依存したコードになっている．その意味で，`vertex.compute` は関数というよりは手続き的である．
- **明示的なメッセージ送受信**　`vertex.compute` の振る舞いを理解するためには，ひとつ前の SS においてどのような意味のデータをどの頂点に送信したかを把握する必要がある．そのため，SS における処理が大きく複雑になればなるほど，プログラムの流れを把握し理解することが難しくなってしまう．

手続き的なコードの更なる問題点を見るために，到達可能な頂点が十分な数 (たとえば 100 個) 見つかったら計算を止めるような問題を考える．以後この問題を「100 到達可能性問題」と呼ぶ．この問題については，更に計算を進めるか計算を終了させるかを決定するために，各 SS において現段階で到達可能と判明した頂点の個数を数える必要がある．このような大域的な情報を得るために，Pregel は「集約」という機構を提供している．集約では，すべての実行中頂点からデータを収集し，和，最大値等の演算を用いてそれらを集計する．集約の結果は，次の SS において実行中の全頂点が利用することができる．

`true` とマークされた頂点の数を数えるために集約を用いると，100 到達可能性問題を解くプログラムは図 3 に示すようになる．ただし簡単のため，到達可能な頂点数は必ず 100 以上あるものとする．ここで，停止するかの判断 (12 行目) の前に集約を行わなければならないため，奇数 SS (7 行目) において集約を行い，偶数 SS において判断を行うようにしている．また `newrch` の値は，奇数 SS で設定され (8 行目)，次の偶数 SS で参照される (17 行目)．そのため，`newrch` は図 1 では局所変数であったが，図 3 では頂点のメンバ変数に変更する必要がある．この例から，次のような更なる問題点を見ることができる．

- **明示的な状態制御**．大域的な情報を得るために集約を用いる場合，異なる SS で異なる振る舞いをするように，SS の偶奇を判断するなど，状態の制御 (状態遷移) を明示的に行わなければなら

```
 1 vertex.compute(v, messages) {
 2  if(superstep == 0) {
 3    v.rch = v.vid == 0;
 4    if(v.rch) sendToNeighbors(v.rch);
 5    else voteToHalt();
 6  } else {
 7    if(superstep % 2 == 1) {
 8      v.newrch = v.rch || or(messages)
 9      aggregate("cnt", v.newrch ? 1 : 0);
10    } else {
11      cnt = read_agg("cnt");
12      if(cnt > 100) {
13        voteToHalt();
14        return;
15      }
16      if(v.newrch != v.rch) {
17        v.rch = v.newrch;
18        sendToNeighbors(v.newrch);
19      }
20    }
21  }
22 }
```

図 3　Pregel-like code for 100-reachability problem.

ない．

図 3 のプログラムは，`voteToHalt()` を注意深く用いている．`true` とマークされた全ての頂点は，集約に参加するために実行中でなければならない一方で，最終的には計算全体を終了させるために，`voteToHalt()` を呼んで停止中に移行しなければならない．しかし，もし 12–15 行目の if 文と 16–19 行目の if 文の順番を入れかえてしまうと，100 到達可能性問題を解くプログラムではなくなってしまう．このことから，次の問題点を見てとることができる．

- **明示的な計算停止制御**．頂点を明示的に停止中に移行させる `voteToHalt()` を適切に用いて全体の計算を適切に終了させるのは，計算処理の順番を注意深く吟味する必要があり，見た目ほど易しいことではない．

これらの問題点を解決するために，我々は，頂点計算を「隣接頂点のデータを送信される受動的な計算」ではなく,「隣接頂点のデータを自ら見にいく能動的な計算」として捉える．このことによって，頂点におけるプログラムは，副作用に基づく手続き的な計算としてではなく,「関数」として捉えることができるようになる．我々は，このような考えに基づいた，頂点中心計算の関数的なモデルを構築し，関数型領域特化言語 Fregel を設計・実装した．Fregel を用いると，頂点主体の計算を宣言的に記述することができる．Fregel プログラムは，既存のグラフ並列処理フレームワーク (現状では Giraph) プログラムへとコンパイルされる．

本稿の貢献点は，次のようにまとめられる．

Peek ベースの頂点主体の関数的モデル　我々は，Pregel の計算を，動的計画法に対応した構造的再帰計算に基づく高階関数として抽象化，定式化する (2 節)．従来の頂点主体計算のモデルでは，頂点は別頂点へ必要なデータを送信しそのデータは別頂点における次の SS での計算で利用される．それとは対照的に我々のモデルは，各頂点は現在の SS の計算で必要な別頂点におけるデータを自ら見にいくような「Peek ベース」のモデルである．

頂点主体のグラフ処理を記述するための関数的領域特化言語　上述の関数的モデルに基づき，我々は宣言的な記述が可能な，純関数的で副作用のない領域特化言語 (以下 DSL と呼ぶ) Fregel を提案する．Fregel では，集約と通信を内包表記を用いて統一的に記述することができる．さらに，4 つの高階関数群を提供し，簡便で合成的なスタイルでのグラフ計算の記述を可能とする．Fregel は Haskell プログラムとしても実行可能なので，Fregel プログラムの開発・デバッグに Haskell 処理系が提供するツール (GHCi 等) を利用することができる．

実用的な性能を達成する実装　我々は，Fregel プログラムを Pregel のオープンソース実装である Graiph プログラムにコンパイルする方法を示す．コンパイルの過程は，まず，グラフ高階関数を複数回使うような Fregel プログラムを平坦化し，グラフ高階関数が一つだけの正規形へと変換する．その後，正規化されたプログラムに対応する Giraph コードを生成する．このような平坦化は，Giraph による計算を複数回立ち上げるのを避け，実用的な実行性能を達成するために重要である．我々はこのようなコンパイラを実装し，いくつかの例題に対して実験を行った．

本稿の構成は，次の通りである．まず，第 2 節で

Peek ベースのモデルの導入を行い，続く第 3 節でそのモデルに基づく関数型領域特化言語 Fregel を導入する．第 4 節で高階関数の複数使用の平坦化を行う正規化を導入し，第 5 節で正規化後の Fregel プログラムの Giraph コードへのコンパイルを述べる．実装されたプロトタイプコンパイラを用いた評価実験の結果を第 6 節で示し，第 7 節で本稿をまとめる．

2 Pregel の関数的モデル

我々は，Pregel における頂点主体の計算を高階関数を用いてモデル化し，そのモデルに基づいた関数型 DSL である Fregel を設計し実装した．ここでは，関数型言語 Haskell の記法を用いて，我々のモデルを説明する．

本稿では，Pregel に以下の制約を課して考える．

- 各頂点における計算の結果，グラフの形は変化しない．
- 頂点間の通信は，有向辺を通じて隣接頂点間でのみ行われる．

後者の制限を導入したモデルは GAS (Gather-Apply-Scatter) モデルと呼ばれ，現実的なモデルであると考えられている [2] [6] [9]．この制限下では，ポインタジャンピング (pointer jumping) を利用するようなアルゴリズムを表現できないものの，本稿では Pregel の基本的なモデルを構築するために，とりあえずこの制限を置いて考える．

2.1 データ型の定義

はじめに，我々の関数的モデルのために必要なデータ型を定義する．`Graph a b` を，頂点の型が `Vertex a b`，入力辺の型が `Edge a b` であるようなグラフを表現する型とする．具体的には，`Graph a b` は，`Vertex a b` 型の頂点のリストである．`Vertex a b` 型の各頂点は，`Int` 型の頂点番号，`a` 型の値，および `Edge a b` 型の入力辺のリストからなる．`Edge a b` 型の入力辺は，辺に付随する `b` 型の値と，その辺で繋がる隣接頂点の組である．

```
data Vertex a b =
  Vertex { vid :: Int, val :: a,
           is :: [Edge a b] }
type Edge a b = (b, Vertex a b)
type Graph a b = [Vertex a b]
```

たとえば，図 2 の右下のグラフ (SS-2 の終わり) は，次のような循環的なデータ構造として表現される．ここで，`v0`，`v1`，`v2`，`v3`，`v4` は，それぞれ左上，右上，左下，中央，右下の頂点を表すものとし，すべての辺は値 1 を持つものとする．

```
g :: Graph Bool Int
g = let v0 = Vertex 0 True []
        v1 = Vertex 1 True [(1,v0)]
        v2 = Vertex 2 True [(1,v0)]
        v3 = Vertex 3 True
               [(1,v1),(1,v2),(1,v4)]
        v4 = Vertex 4 False []
    in [v0, v1, v2, v3, v4]
```

2.2 関数的モデル

頂点主体の並列計算では，各頂点は以下のような処理を繰返し実行する．

1. 入力辺で接続されている隣接頂点から，1 回前の繰返し処理において計算されたデータを受信する．
2. 受信したデータと，自身の 1 回前の繰返し処理における計算結果のデータを用いて，問題に応じた計算を行う．その計算の過程において必要があれば，集約を用いて大域的な情報を得る．
3. 出力辺で接続されている全隣接頂点に，2 における計算結果のデータを送信する．各隣接頂点は，このデータを次の繰返し処理で受け取る．

上で述べた繰返し処理 1〜3 の 1 回分を，本稿では logical superstep (以下 LSS) と呼ぶ．LSS には集約が含まれ得るので，LSS 1 回分の処理が実際には Pregel における複数の SS で実現される可能性がある．

我々は，LSS を関数として記述し，これを「LSS 関数」と呼ぶ．後述するように，LSS 関数ではデータの送受信を明示的には記述せず，かわりに自身の再帰呼出しを用いる．LSS 関数に与えられる引数は，LSS 関数の何回目の繰返しかを表す整数値 (以下「クロック」と呼ぶ) と，LSS 関数を実行する頂点の二つであ

る．したがって，LSS 関数の型は，結果の型を r とすると Int -> Vertex a b -> r と表される．

lss を LSS 関数とすると，lss 0 v は頂点 v における計算の初期値を，lss t v (ただし t $>$ 0) はクロック t において頂点 v が行う計算を表す．ここでクロック t における計算では，クロック t-1 における自身の値 (すなわち lss (t-1) v)，および，入力辺で結ばれている各隣接頂点 u における値 (すなわち lss (t-1) u) を用いる．このような LSS 関数は，以下の二つの関数を用いて特徴付けられる．ひとつは t $=$ 0 における振る舞いを規定する「初期化関数」，もうひとつは t $>$ 0 における振る舞いを規定する「ステップ関数」である．これらの二つの関数を用いれば，LSS 関数の一般形 lssGen は次のような関数として定義することができる．ここで，f0 と ft はそれぞれ初期化関数とステップ関数である．

```
lssGen :: (Vertex a b -> r) ->
      (r -> [(b,r)] -> Vertex a b -> r) ->
      Int -> Vertex a b -> r
lssGen f0 ft 0 v = f0 v
lssGen f0 ft t v =
  ft (lssGen f0 ft (t - 1) v)
     [(e, lssGen f0 ft (t - 1) u) |
                          (e,u) <- is v]
     v
```

個々の問題に対する LSS 関数は，lssGen に対して適切な初期化関数とステップ関数 (init と step とする) を次のように実引数として与えることによって得ることができる．

```
lss = lssGen init step
```

g $=$ [v_0, v_1, v_2, ...] を Pregel における処理対象のグラフとする．クロック t における全頂点での LSS 関数 lss の計算結果のリストは [lss t v_0, lss t v_1, lss t v_2, ...] と表される．makeGraph g [r_0, r_1, r_2, ...] は，g と同じ形をして，i 番目の頂点は値 r_i を，辺は g のそれと同じ値を持つようなグラフを返すものとする．すると，次の無限リスト

```
[makeGraph g [lss 0 v0, lss 0 v1, ...],
 makeGraph g [lss 1 v0, lss 1 v1, ...],
 makeGraph g [lss 2 v0, lss 2 v1, ...],
 ...]
```

は，LSS 関数 lss の無限回の繰返しを表現する．ここで，この無限リストの t 番目の要素は，クロック t における全頂点の LSS 関数の結果から構成されるグラフである．このような無限リストは，初期化関数，ステップ関数，処理対象のグラフを引数としてとる高階関数 pregelIter によって生成できる．

```
pregelIter :: (Vertex a b -> r) ->
     (r -> [(b,r)] -> Vertex a b -> r) ->
     Graph a b -> [Graph r b]
pregelIter f0 ft g =
  map (\t -> makeGraph g
               (map (lssGen f0 ft t) g))
      [0..]
```

pregelIter は無限リストを返すように定義されているが，実際には，無限に計算を続けるのではなく，どこか適切な時点で計算を打ち切って最終結果を得る必要がある．打ち切り条件の典型例のひとつは，計算結果が変化しなくなり定常状態に至った時点である．fixedValue :: (Eq r, Eq b) => [Graph r b] -> Graph r b を，上のような無限リストから定常状態のグラフを返す関数とする．すると，fixedValue (pregelIter init step g) により，定常状態のグラフを最終結果として得ることができる．

別の打ち切り条件の例としては，lss の結果のグラフがある条件を満足するに至った場合がある．untilValue :: (Graph r b -> Bool) -> [Graph r b] -> Graph r b を，条件が満足されたかを判定する述語となる関数とグラフの無限リストを与えられ，その述語を満足する最初のグラフを返す関数とする．すると，untilValue pred (pregelIter init step g) により，pred を満足するようなグラフを結果として得ることができる．

さらに，LSS 関数を指定回数繰り返した上で計算を打ち切るような打ち切り条件も考えられる．

最後に，関数 pregelModel を，打ち切り条件を規定する関数と pregelIter の合成として定義する．

```
reInit :: Vertex a b -> Bool
reInit v = vid v == 0

reStep :: Bool -> [(b,Bool)] ->
          Vertex a b -> Bool
reStep p eqs v =
  p || or [ q | (e,q) <- eqs ]

reAllPregelModel ::
  Graph a b -> Graph Bool b
reAllPregelModel =
  pregelModel reInit reStep fixedValue

re100PregelModel ::
  Graph a b -> Graph Bool b
re100PregelModel =
  pregelModel reInit reStep
    (untilValue ((> 100) . numTrueVertices))
  where numTrueVertices vs =
          length (filter val vs)
```

図 4　関数型モデルによる到達可能性問題の定式化．

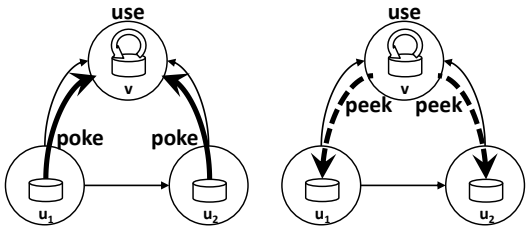
```
pregelModel :: (Vertex a b -> r) ->
      (r -> [(b,r)] -> Vertex a b -> r) ->
      ([Graph r b] -> Graph r b) ->
      Graph a b -> Graph r b
pregelModel f0 ft term =
  term . pregelIter f0 ft
```

我々は，この関数 `pregelModel` が，Pregel における計算を表現するモデルと考える．

2.3　簡単な例題

上で説明した Pregel のモデルに基づいて到達可能性問題を定式化すると，図 4 のようになる．`reAllPregelModel` は全到達可能性問題を，`re100PregelModel` は 100 到達可能性問題を表す．ここで `numTrueVertices` は，与えられたグラフの中で値 `True` を持つ頂点の総数を返す関数とする．両者の違いは，打ち切り条件として全到達可能性問題は `fixedValue` を使うが，100 到達可能性問題は `untilValue` を使うという点だけである．`reInit` と `reStep` によって特徴付けられる LSS 関数には，Pregel コードに見られた `voteToHalt()` に相当する処理が一切含まれない点に注意されたい．

図 4 の定義は Haskell の関数であり，Haskell 処理系を利用してそのまま実行できる．しかし，`reAllPregelModel`，`re100PregelModel` は同一の引数に対する LSS 関数の呼び出しを多数回重複して行うので，効率的ではない．そこで，我々のモデルにおいては，重複した呼び出しを回避するメモ化機構の存在を，暗黙のうちに仮定するものとする．



(a) 元のモデル　　(b) 我々のモデル

図 5　データ送受信の捉え方の違い．

2.4　モデルの特徴

`lssGen` を用いた LSS 関数は，入力グラフの構造を基礎とする一種の構造的再帰 (structural recursion) の形をしている．グラフは全体としては循環的な構造をしているが，LSS 関数の再帰呼び出しが無限再帰に陥らないのは，常にひとつ前のクロック値 (`t-1`) を用いて再帰呼び出しを行っているためである．加えて，上で述べたようなメモ化機構を前提として考えると，クロック値を減らすような LSS 関数の再帰実行は，グラフ上の動的計画法として捉えることができる．

LSS 関数の中には，隣接頂点間のデータの送受信を明示的には記述しない．データ送受信の捉え方を，元の Pregel モデルと我々の関数的モデルで比較すると，元の Pregel は データ送信を明示的に記述する「explicit-poking」のスタイルであるのに対し，我々のモデルはステップ関数の実引数を通して送信されたデータを見るという「implicit-peeking」のスタイルである．この違いを，図 5 に示す．

以上をまとめると，我々のモデルは 1 節で述べた問題を次のように解決している．

- 第 1 の「副作用に依存したコード」の問題については，我々のモデルは，各頂点で繰返し同期的に実行される計算を，副作用のない純関数的な LSS 関数として定義することで解決している．LSS 関数は，グラフの構造に基づいた構造的再

帰の形をしており，その再帰的実行はメモ化機構を前提としたグラフ上の動的計画法として解釈することができる．

- 第 2 の「明示的なメッセージ送受信」の問題については，LSS 関数は頂点間の明示的なデータ送受信を記述せず，そのかわり，隣接頂点に関する LSS 関数の再帰呼び出しを用いることで解決している．この再帰呼び出しは，implicit-peeking 方式の通信として捉えることができる．
- 第 3 の「明示的な状態制御」の問題については，LSS 関数は，BSP モデルのバリア同期によって複数の小さな SS に細分化されるような処理を，細分化せずにまとめて記述できるようにすることで解決している．
- 第 4 の「明示的な計算停止制御」の問題については，グラフに対する計算全体を，各クロックにおける計算結果のグラフを順に並べた無限リストとして表現し，そのリストから望みのグラフを適切に取り出す関数を外から与えることで解決している．その結果，LSS 関数には計算の停止に関する記述が一切含まれない．

3 Fregel: 大規模グラフ処理のための関数型 DSL

3.1 Fregel の特徴

Fregel は大規模グラフ処理のための関数型領域特化言語である．Fregel はデータへのアクセス，データの集約，および通信を関数型言語の見地から抽象化している．これは，2 節で導入した `pregelModel` に基づき設計されている．さらに，様々なグラフ処理を簡便に記述するための 4 つの高階関数を提供している．

まず，Fregel では頂点のデータには頂点で添え字づけられたテーブルによってアクセスできる．***prev*** テーブルは直前（$t-1$）のクロックでの頂点データを保持している．技術的な理由から，もうひとつ ***curr*** テーブルというものも用いているが本稿では深入りしない．これらのテーブルは，2 節で述べたメモ化を明示的に実現している．これにより，プログラマはデータへのアクセスをより直接的に記述できる．

次に, Fregel では通信と集約を，特定の生成子（generator）を伴う内包表記で記述する．集約は，グラフ中の全ての頂点からなる生成子で記述する．隣接頂点との通信には，それら頂点のみからなる生成子を用いる．

さらに， Fregel は以下の 4 つの高階関数を提供している．関数 ***fregel*** は 2 節で導入した `pregelModel` に対応する．関数 ***gzip*** は 2 つの同じ形状のグラフを取り，対応するそれぞれの頂点を組にする．関数 ***gmap*** はグラフの各頂点に与えられた関数を適用する．関数 ***giter*** は繰り返し計算を提供する．

Fregel の構文は Haskell のそれに従っており，Fregel プログラムは Haskell の処理系（例えば GHCi）の上で実行できる．そのため，テストやデバッグを Haskell 処理系で行い，その後に Pregel プログラムへコンパイルし大規模グラフ処理を行うことができる．

3.2 Fregel の構文

Fregel において，頂点は 2 種類の辺のリストを持つ．ひとつは通常の意味での incoming 辺，もうひとつは辺を反転させた際の incoming 辺である．以降では,「逆向きの辺」は後者のリストの要素を指す．後者は 2.1 節で考えたモデルには現れなかったが，プログラマの利便性のため全ての頂点がこれをもつようにした．

Fregel の構文の主要な部分を図 6 に示す．太斜体の語句は Fregel における重要な予約語である．

Fregel プログラムはグラフからグラフへの関数の定義からなる．ひとつの LSS 関数は初期化関数とステップ関数のふたつで定義される．LSS 関数は複数の値を返すことも多いため，プログラマはレコードを用いてこれを表し，従って各頂点がレコードを保持することになる．Fregel はレコードアクセスの記法を提供している．

Fregel の式は Haskell の標準的なものに従っている．生成子を伴う内包表記，フィールド選択演算子 (.^) を頂点 v に適用することによるテーブルからの値取り出し，そして ***fregel*** 等の高階関数によるグラフ処理式などが含まれる．生成子は 3 種類ある．グラフの全ての頂点からなるものと，頂点 v の隣接頂点及びおよびその辺の値からなるもの（***is*** v は通常の

$$
\begin{array}{lll}
prog & := & f\ g\ =\ e \\
e & := & \textbf{let}\ decl_1\ \cdots\ decl_n\ \textbf{in}\ e \\
 & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & | & f\ e_1\ \cdots\ e_n \\
 & | & comb\ [\ e\ |\ gen,\ e_1,\ \ldots,\ e_n\ ] \\
 & | & table\ v\ .\hat{}\ fld_1\ .\hat{}\ \cdots\ .\hat{}\ fld_n \\
 & | & \textbf{fregel}\ f_1\ f_2\ tc\ g \\
 & | & \textbf{gmap}\ f\ g \\
 & | & \textbf{gzip}\ g_1\ g_2 \\
 & | & \textbf{giter}\ f_1\ f_2\ tc\ g \\
decl & := & f\ x_1\ \cdots\ x_n\ =\ e \\
 & | & f\ v\ \textbf{prev}\ \textbf{curr}\ =\ e \\
gen & := & u\ \leftarrow\ g\ |\ (ed,u)\ \leftarrow\ \textbf{is}\ v\ |\ (ed,u)\ \leftarrow\ \textbf{rs}\ v \\
table & := & \textbf{prev}\ |\ \textbf{curr}\ |\ \textbf{val} \\
tc & := & \textbf{Fix}\ |\ \textbf{Until}\ (\lambda\ g.e)\ |\ \textbf{Iter}\ e \\
f & := & \text{variable representing a function / operator} \\
g & := & \text{variable representing a graph} \\
u,v & := & \text{variable representing a vertex} \\
ed & := & \text{variable representing an edge} \\
fld & := & \text{field name} \\
comb & := & \texttt{max}\ |\ \texttt{or}\ |\ \cdots
\end{array}
$$

図 6　Fregel の主要な構文

辺，**rs** v は逆向きの辺に対応する）である．

変数・関数定義は 2 種類に分類される．ひとつは普通の関数や定数を定義するものである．LSS 関数のための初期化関数もこれで定義する．ステップ関数は特殊な形式に従うもので，頂点 v に加え **prev** と **curr** の 2 つのテーブルを引数に取る．なお，**prev** と **curr** に加え，入力グラフの値にアクセスするためのテーブルとして **val** が提供されている．高階関数 **fregel** と **giter** のための停止条件としては **Fix**, **Until**, **Iter** の 3 つが用意されている．**Fix** は不動点を表す．**Until** は関数で停止条件を指定する．**Iter** は LSS 関数を指定された回数適用する．

3.3　例: 到達可能性問題

Fregel プログラムの例として，図 7(a) に全到達可能性問題を解くものを示す．LSS 関数は，それを呼んだ頂点が現時点で到達可能であるかどうかを示す真偽値を返す．そのため，`rch` フィールドにこの真偽値を保持するレコード `RVal` を用いる．

関数 `reAll` が主要な部分であり，LSS 関数の初期化関数とステップ関数を定義している．初期化関数 `init` は，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．なお，

```
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 =
6              prev v .^ rch ||
7               or [prev u .^ rch | (e, u) ← is v]
8        in RVal newrch
9   in fregel init step Fix g
```

(a) 全到達可能性問題のプログラム

```
1  data RVal = RVal { rch :: Bool } deriving Eq
2  re100 g =
3   let init v = RVal (vid v == 0)
4       step v prev curr =
5        let newrch =
6              prev v .^ rch ||
7               or [prev u .^ rch | (e, u) <- is v]
8        in RVal newrch
9   in fregel init step
10      (Until
11       (\g -> sum [1 | u <- g, val u .^ rch]
12              > 100))
13      g
```

(b) 100 到達可能性問題のプログラム

図 7　到達可能性問題のための Fregel プログラム

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，直前のクロックでの結果を隣接頂点から収集する．これは **is** `v` を生成子とする内包表記で実現される．それぞれの隣接頂点 `u` の直前のクロックの結果は **prev** `u` から `rch` フィールドを取得し，それらの論理和を `or` 関数で求めている．

定義された `init` と `step` は **fregel** の引数となる．第 3 引数の **Fix** は停止条件を与える．第 4 引数は入力グラフである．

図 7(b) は 100 到達可能性問題のための Fregel プログラムである．このプログラムは図 7(a) のものとほとんど同じである，停止条件のみが異なる．ここでは停止条件に **Until**（これは関数モデルでの `untilValue` に対応する）を用いている．**Until** は条件を指定する関数を引数に取るが，この関数では到達可能な頂点を集約によって計算している．

3.4　例：強連結成分分解

Fregel の特徴であるグラフ上の高階関数を複数使う例として，与えられたグラフの強連結成分分解を行う Fregel プログラムを Fig. 8 に示す．この計算は，

入力グラフに対し，互いに行き来できる頂点集合（強連結成分）毎に固有の番号を付加する計算である．このプログラムは，Yan ら [12] の min-label アルゴリズムに基づく．

このプログラムは次のよっつの計算段階をすべての頂点が自身の固有番号を決定するまで繰り返す．

(1) 初期化．まだ固有番号が決まっていない頂点は *notfound* フラグをセットする．これは即ち，この頂点を以降の計算に巻き込むことを意味する．

(2) 前進伝播．*notfound* をセットした頂点は，その *minv* をまずは自身の頂点番号とし，以降は自分に入ってくる辺の先の頂点の *minv* の最小値で自身の *minv* を更新することを繰り返す．

(3) 後進伝播．前進伝播と同様に，ただし自身から出て行く辺の先の頂点の *minv* で更新を行う．

(4) 固有番号判定．*notfound* をセットした頂点のうち，その前進伝播と後進伝播の結果が同じであったものは，その値を自身の固有番号として決定する．

図 8 のプログラムはネストした繰り返し構造をもつ．外側の繰り返しは ***giter*** で記述され，上記の (1) – (4) の繰り返し適用を表現している．この繰り返しでは，各々の頂点は *sccId* フィールドを持つレコード *C* をもつ．このフィールドは -1（未定）で初期化され，計算の進行に伴い頂点の固有番号となる．

(1) – (4) の計算の最中は，基本的に各頂点はふたつのフィールド *minv* と *notfound* をもつ *MN* レコードをもつ．それぞれの意味は上記のとおりである．初期化の処理は ***gmap*** を用いて実装され，結果が中間グラフである *ga* に束縛される．その後，前進伝播と後進伝播の計算のためにふたつの内側の繰り返しが ***fregel*** で記述されている．これらはともに *ga* を入力として計算を行う．これらの結果は *gf* と *gb* に束縛され，***gzip*** で組化されたあとに固有番号判定の処理を ***gmap*** によって適用される．

一般に，このような複数の計算段階やネストした繰り返しを持った計算を，単一の頂点計算での記述を強いる Pregel プログラムとして実現することは難しい．それに対し，Fregel では，提供される高階関数の組合せてとしてこれらの計算を簡潔に記述することが出来る．

4 Fregel プログラムの正規化

Fregel プログラムはグラフ上の高階関数の使用を複数含むことが出来る．このような Fregel プログラムの愚直なコンパイルは個々の高階関数を 1 回の Pregel 計算の実行に対応させることであるが，Pregel の大きなスタートアップコストによりこの方法でコンパイルされたプログラムは非効率的である．そこで，我々は，高階関数の複数の使用を含む Fregel プログラムを，***fregel*** をたった一回だけ使用する Fregel プログラムへと変換する正規化を導入する．

正規化されたプログラムは，停止条件を ***Fix*** とした ***fregel*** を 1 回だけ使用する次の形のプログラムである．

```
1  prog g = let (bindings)...
2           in fregel init step Fix g
```

正規化の直感は，元のプログラムの動作を **模倣する** ステップ関数を構築することである．即ち，元のプログラムのどの高階関数の計算を実行中であるか，その計算が終わったら次はどの高階関数の計算に遷移するか，という情報を持った状態遷移機械を構築することに相当する．この状態遷移機械は，元のプログラムにおける（中間）グラフ全てを組化したグラフを対象に計算を行う．

元のプログラムのどの高階関数の計算を実行中であるかをフェーズと呼ぶことにする．各フェーズでは，元のプログラムの高階関数で使われていたステップ関数を実行する．

例として，図 8 の *scc* を正規化したプログラムを図 9 に示す．

元のプログラム *scc* は中間グラフとして *gr*, *ga*, *gf*, *gb*, *gfb*, *g'* を含む．よって，これらの計算を状態としてもつ状態遷移機械が正規化後のプログラムのステップ関数である．その状態遷移の様子は図 10 に示されるとおりである．この状態遷移の関係は，正規化後のプログラムでは *stay* と *next* として実現されている．

正規化後のプログラムは，元のプログラムにおける中間グラフをすべて組化したグラフを計算対象とす

```
1  data MN = MN { minv :: Int, notfound :: Bool } deriving Eq
2  data C = C { sccId :: Int } deriving Eq
3  scc g =
4    let f_init v = if val v .^ sccId < 0 then MN (vid v) True else MN (val v .^ sccId) False
5        f_fw v prev curr =
6          let c' = (prev v .^ minv) `min` minimum [ prev u .^ minv | (e,u) <- is v, prev u .^ notfound ]
7          in if prev v .^ notfound then MN c' (prev v .^ notfound) else prev v
8        f_bw v prev curr =
9          let c' = (prev v .^ minv) `min` minimum [ prev u .^ minv | (e,u) <- rs v, prev u .^ notfound ]
10         in if prev v .^ notfound then MN c' (prev v .^ notfound) else prev v
11       detect v = if val v .^ _fst .^ minv == val v .^ _snd .^ minv then C (val v .^ _fst .^ minv) else
               C (-1)
12       f0 v = val v
13       sccInner0 v = C (-1)
14       sccInner g = let ga = gmap f_init g
15                        gf = fregel f0 f_fw Fix ga
16                        gb = fregel f0 f_bw Fix ga
17                        gfb = gzip gf gb
18                        g' = gmap detect gfb
19                    in g'
20       gr = giter sccInner0 sccInner Fix g
21   in gr
```

図 8　Fregel program for solving strongly-connected components problem.

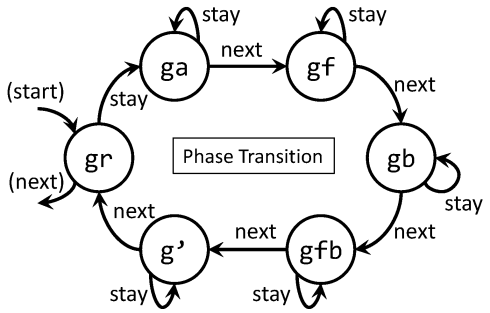


図 10　scc の正規化のための状態遷移機械．

る．そのため，正規化後のプログラムで使用されるレコード *ND* は，元のプログラムで使われるレコードをフィールドとしてもつ構造となっている．さらに，どの状態の計算を行っているのかを示す *phase* フィールドと，各状態内での LSS をカウントするフィールド *ss* をもつ．さらに，***giter*** が使用されていた場合には，その ***giter*** 毎の繰り返しカウンタをもつ．

詳細な正規化アルゴリズムは我々の ICFP 論文[4]を参照されたい．

最後に，正規化における最適化の余地について述べておく．上記の正規化では，単純のために ***gzip*** による組化も他の ***gmap*** 等の計算と同様に扱っている（即ち，組を作るという単純な計算だけを行う状態が存在する）．しかし，全ての中間グラフが組化されているという正規化後のモデルでは，そもそも個々の組化にあまり意味が無い．そのため，***gzip*** による組化をキャンセルするように正規化を行うことで，多少の性能向上が期待できる．もうひとつの単純な最適化としては，独立なフェーズの同時計算が考えられる．BSP モデルでは，SS の数を減らすことが大きな性能向上の鍵となる．例えば，上記の強連結成分分解のプログラムでは，*gf* と *gb* の計算は独立しているため，同時に実行することができる．*gf* と *gb* とがそれぞれ ss_{gf} 回と ss_{gb} 回の SS を必要としていたならば，個々のフェーズとして計算した場合には $ss_{gf} + ss_{gb}$ 回の SS が必要であるのに対し，同時計算の最適化後は $\max(ss_{gf}, ss_{gb})$ 回の SS で済むことになり性能向上が期待できる．

5　Fregel プログラムの Giraph コードへのコンパイル

我々は オープンソース Pregel フレームワークである Giraph [1] で動作するコードへのプロトタイプコンパイラを実装した．以下，正規化後の Fregel プログラムのコンパイルを簡単に述べる．詳細なコンパイル方法は我々の ICFP 論文[4] を参照されたい．このコンパイルには Giraph 特有の機能は用いておらず，ゆえに任意の Pregel フレームワークへも同様にコン

```
data ND = ND { phase :: Int, ss :: Int,
               dat_gr :: C, dat_ga :: MN, dat_gf :: MN, dat_gb :: MN, dat_gfb :: Pair MN MN, dat_g' ::
                    C,
               ss_gr :: Int} deriving Eq
scc g =
 let step v prev curr =
      let d_gr = if prev v .^ phase == 1
                 then if prev v .^ ss_gr == 0 then C (-1) else prev v .^ dat_g'
                 else prev v .^ dat_gr
          d_ga = if prev v .^ phase == 2
                 then if prev v .^ dat_gr .^ sccId < 0 then MN (vid v) True else MN (prev v .^ dat_gr
                      .^ sccId) False
                 else prev v .^ dat_ga
          d_gf = if prev v .^ phase == 3
                 then if prev v .^ ss == 0 then prev v .^ dat_ga
                      else
                       let c' = (prev v .^ dat_gf .^ minv) `min`
                                minimum [ prev u .^ dat_gf .^ minv | (e,u) <- is v, prev u .^ dat_gf
                                     .^ notfound ]
                       in if prev v .^ dat_gf .^ notfound then MN c' (prev v .^ dat_gf .^ notfound)
                            else prev v .^ dat_gf
                 else prev v .^ dat_gf
          d_gb = if prev v .^ phase == 4
                 then if prev v .^ ss == 0 then prev v .^ dat_ga
                      else
                       let c' = (prev v .^ dat_gb .^ minv) `min`
                                minimum [ prev u .^ dat_gb .^ minv | (e,u) <- rs v, prev u .^ dat_gb
                                     .^ notfound ]
                       in if prev v .^ dat_gb .^ notfound then MN c' (prev v .^ dat_gb .^ notfound)
                            else prev v .^ dat_gb
                 else prev v .^ dat_gb
          d_gfb = if prev v .^ phase == 5 then Pair (prev v .^ dat_gf) (prev v .^ dat_gb) else prev v
                .^ dat_gfb
          d_g' = if prev v .^ phase == 6
                 then if prev v .^ dat_gfb .^ _fst .^ minv == prev v .^ dat_gfb .^ _snd .^ minv
                      then C (prev v .^ dat_gfb .^ _fst .^ minv) else C (-1)
                 else prev v .^ dat_g'
          pend = (prev v .^ phase == 1 && prev v .^ ss_gr > 0 && and [ prev u .^ dat_gr == curr u .^
               dat_gr | u <- g ])
                   || (prev v .^ phase == 2 && True )
                   || (prev v .^ phase == 3 && prev v .^ ss > 0 && and [ prev u .^ dat_gf == curr u .^
                        dat_gf | u <- g ])
                   || (prev v .^ phase == 4 && prev v .^ ss > 0 && and [ prev u .^ dat_gb == curr u .^
                        dat_gb | u <- g ])
                   || (prev v .^ phase == 5 && True ) || (prev v .^ phase == 6 && True )
          phase' = if pend then next (prev v .^ phase) else stay (prev v .^ phase)
          ss' = if prev v .^ phase  > 0 && prev v .^ phase == curr v .^ phase then prev v .^ ss + 1
                else 0
          ss_gr' = if prev v .^ phase == 1 then if pend then 0 else prev v .^ ss_gr + 1 else prev v .^
                    ss_gr
       in ND phase' ss' d_gr d_ga d_gf d_gb d_gfb d_g' ss_gr'
      init v = ND 1 0 (C 0) (MN 0 False) (MN 0 False) (MN 0 False) (Pair (MN 0 False) (MN 0 False)) (C
           0) 0
 in fregel init step Fix g
```

図 9　図 8 の `scc` **を正規化したプログラム．** `stay` **と** `next` **の実際定義は略しているが，等価な定義は次のとおりである：**

```
stay i = if i == 1 then 2 else i, next i = if i == 1 then 0 else if i == 6 then 1 else i + 1.
```

パイル可能である．コンパイル全体の流れは図 11 に示される．

正規化後のプログラムのコンパイルは次の 5 段階でコンパイルされる．

1. 実行順序の整理
2. ステップ関数の SS への分解
3. 頂点フィールドの列挙
4. 集約の列挙と通信への変換

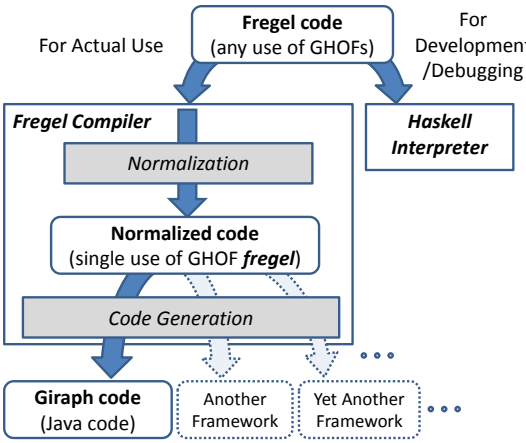


図 11　Fregel プログラムのコンパイルとインタープリテーション (GHOF: グラフ高階関数).

5. コード生成

コンパイルの最初の段階は，定義の依存関係の整理である．Fregel プログラムは宣言的であるため，命令的な Giraph プログラムへの変換のために依存関係に基づいた実行順序の決定を行う．

次に，ステップ関数を複数の SS に分解する．Fregel プログラムにおいては通信が内包表記で表現されている．Pregel の採用する BSP モデルの制限から，通信結果は次の SS でないと利用できない．すなわち，内包表記の結果を別の内包表記で用いる場合などには，ステップ関数を複数の SS に分割する必要が生じる．

第三段階として，頂点のフィールド（SS をまたいで保存すべき変数）を列挙する．そのようなフィールドは次のような分類となる．上記 4 つは正規化後の Fregel プログラムの頂点がもつレコード型に対応する．最後の要素はステップ関数の分割で導入されるものである．

- フェーズ (`v.phase`)
- フェーズ内 SS のカウンタ (`v.ss`)
- 個々の高階関数 ***giter*** の為の繰り返しカウンタ
- Fregel プログラムでの頂点レコードのフィールド
- ステップ関数内の複数 SS への分割で生じた SS をまたぐ局所変数

次の段階として，内包表記の通信への変換を行う．主な処理は次のふたつである．

- 入力辺 ***is*** と逆向き辺 ***rs*** が生成子として用いられている場合，これは通常の頂点間通信に変換される．より具体的には，送り手の前 SS にメッセージ送信を追加し，受け手の現 SS にメッセージの処理を追加する．
- グラフ全体が生成子として用いられている場合，これは集約へと変換される．より具体的には，すべての頂点で前 SS に集約への値投下を追加し，現 SS に集約の読出を追加する．

これの他，多くの Pregel フレームワークの実装がメッセージ用のクラスの定義を必要とするため，全ての内包表記の処理の後に必要なメッセージ型を内包するクラスを生成する．

最後に，Giraph コードのための `vertex.compute` 関数を生成する．生成される関数の大枠は次のとおりである．この段階までに処理されなかった Fregel コードは単純な式のみであり，それらは単純な Java コードにコンパイル可能である．

```
1  public void compute(Vertex<...> vertex,
2                      Iterable<MyMessage> msgs) {
3   MyVertex v = vertex.getValue();
4   if (getSuperstep() == 0) {
5    // initializing phase and step
6    v.phase = new IntWritable(0);
7    v.ss = new IntWritable(0);
8   }
9   switch (v.phase.get()) {
10  case 0: {
11   switch (v.ss.get()) {
12   case 0: {
13    // code for init-function for phase 0
14    v.ss = new IntWritable(1);
15   } break;
16   case 1: {
17    // code for judging termination
18    if (...) { vertex.voteToHalt(); return; }
19    // code for phase transition
20    if (...) { v.phase = new IntWritable(1);
21          return; }
22    ...
23   } break;
24   case 2: {
25    // handling (received) messages
26    for (MyMessage msg : msgs) { ... }
27    // computation in LSS function
28    ...
29    // sending message for next superstep
30    // update v.ss
31   } break;
32   ...
33  }
34  vertex.setValue(v);
35 }
```

6 評価

6.1 PC クラスタ上での性能とオーバーヘッドの評価

Fregel プログラムからコンパイルされたコードの性能を評価するため，以下の 3 つの問題を対象として評価実験を行った．

- 最短路問題 (図. 7 の到達可能性とほぼ同様)
- 最密部分グラフ問題：この問題に対する逐次プログラムを書くことは容易であるが，Pregel プログラムを書くのは自明ではない [3].
- 強連結成分分解 (第 3.4 節)

データには，2 つのランダム生成したグラフを用いた．“rand-s” は頂点数 $|V| = 1 \times 10^6$ および枝数 $|E| = 1 \times 10^7$ であり，“rand-d” は頂点数 $|V| = 1 \times 10^6$ および枝数 $|E| = 1 \times 10^8$ である．

実験のハードウェア環境は，ギガビットイーサネットで接続された 16 台の PC からなる PC クラスタで，各 PC は Intel Core i5 CPU (うち 9 台は Core i5-2500，残り 7 台は Core i5-760) と 8 GB のメモリ，128 GB の SSD を持つ．OS と Java，Hadoop，Giraph のバージョンはそれぞれ Ubuntu 14.04.3 LTS，1.8.0_66，0.20.203.0，1.2.0-SNAPSHOT である．

表 1 に，各実行に対する実行時間 (頂点の初期配置と出力にかかる時間を除く) と SS 数，総メッセージ量を示す．これらの値は各パラメータに対して 5 回の実験を行った平均値である．P は並列計算に用いたワーカー PC 数である．Fregel からコンパイルしたコードは妥当な時間で実行でき，またワーカー PC 数を増やすことで並列速度向上を得ることができた．

Fregel からコンパイルされたコードのオーバーヘッドの要因として，以下の 4 つを考えた．(1) 頂点の持つデータがより大きい，(2) メッセージ通信量がより多い，(3) superstep 数がより多い，(4) 計算途中で `voteToHalt` を用いていない．最短路問題と最密部分グラフ問題に対して手書きした複数のコードを実行し，Fregel からコンパイルされたコードのオーバーヘッドを解析した．

最短路問題に対しては，以下の 6 つのプログラムを用いた．(a) Pregel の提案論文 [7] で示されているコードとほぼ等価なコード．(b) 生成された頂点クラスを用いること以外は (a) と同じアルゴリズムであるコード．(c) 生成された頂点クラスとメッセージクラスを用いること以外は (a) と同じアルゴリズムであるコード．(d) コンパイルしたコードと同じ superstep 数となるように変更したアルゴリズムによるコード．(e) fixpoint に至るまで `voteToHalt` を用いたいように変更したアルゴリズムによるコード．(f) 各頂点が常に隣接頂点へ向けてメッセージを送るように変更したアルゴリズムによるコード．表 2 に，これらのプログラムに対する実行時間と superstep 数，総メッセージ量を示す．(b) と (c) の差，および，(e) と (f) の差から，Fregel からコンパイルされたコードのオーバーヘッドの主要因が総メッセージ量によるものであることが分かる．また，(c) と (d) の差より，superstep 数がより多いことも速度低下の要因である (今回の場合では 20–40%)．これらのオーバーヘッドと比べると，頂点の持つデータがより大きいことや `voteToHalt` を用いていないことによるオーバーヘッドはとても小さい．

最密部分グラフに対しては，以下の 4 つのプログラムを用いた．(a) superstep 数とメッセージ通信量が最小となるように注意して手書きしたコード †1．(b) 生成された頂点クラスを用いること以外は (a) と同じアルゴリズムであるコード．(c) 生成された頂点クラスとメッセージクラスを用いること以外は (a) と同じアルゴリズムであるコード．(d) コンパイルしたコードと同じ superstep 数となるように変更したアルゴリズムによるコード．表 3 に，これらのプログラムに対する実行時間と superstep 数，総メッセージ量を示す．これらの結果から，総メッセージ量が性能に最も大きな影響があることが分かる．また，(c) と (d) の差である superstep 数の増加や，(d) と Fregel コードの差が無視できない．`double` 型などの primitive 型と `DoubleWritable` などの Hadoop 特有の型との間の変換が非常に多くなってしまっていることがその

†1 このプログラムでは計算途中での `voteToHalt` を用いていない．それを効果的に用いるためには，非アクティブ状態となった全頂点をアクティブにする機構が必要である．

表 1　PC クラスタにおける実験結果

	実行時間 (s)						SS 数	総通信量
	P=1	P=2	P=4	P=8	P=12	P=16		(MB)
最短路問題 (rand-s)	294.8	137.4	63.53	35.33	25.40	26.86	58	4,762
最短路問題 (rand-d)	N/A	N/A	391.8	192.1	150.8	135.1	50	40,810
最密部分グラフ問題 (rand-s)	95.94	52.11	31.17	19.60	16.16	13.91	185	389
強連結成分分解 (rand-s)	404.7	160.0	73.07	41.00	30.53	28.32	58	4,622

表 2　最短路問題に対するオーバーヘッドの解析

	実行時間 (s)		superstep 数	メッセージ量
	P=4	P=16		
a. 手書き	7.69	4.23	29	413
b. + 生成頂点	7.84	4.39	29	413
c. + 生成メッセージ	16.35	6.67	29	413
d. + 2 倍の superstep	19.10	9.32	58	413
e. + VoteToHalt なし	18.62	9.21	58	413
f. + 常に送信	59.98	26.00	58	4,762
Fregel	63.53	26.86	58	4,762

表 3　最密部分グラフ問題に対するオーバーヘッドの解析

	実行時間 (s)		superstep 数	メッセージ量
	P=4	P=16		
a. 手書き	14.12	8.14	125	222
b. + 生成頂点	15.29	8.99	125	222
c. + 生成メッセージ	18.23	9.54	125	389
d. + 1.5 倍の superstep	23.29	13.21	187	389
Fregel	31.17	13.91	185	389

理由であると考える．

6.2　Amazon EMR Cloud 環境での評価実験

Amazon EMR cloud 環境を用いて，より大きな実験を行った．実験に用いた計算ノードは m3.xlarge (CPU Intel Xeon E5-2670 v2, メモリ 15 GB, SSD) であり，Java 1.7，Hadoop 1.0.3，および，Giraph 1.2.0-SNAPSHOT を用いた．

表 4 に，頂点数 $|V| = 5 \times 10^7$ および枝数 $|E| = 5 \times 10^8$ となるようにランダム生成したグラフに対する最短路問題のプログラムの実行時間を示す．Fregel からコンパイルしたコードには (8 倍から 10 倍の) オーバーヘッドがあるものの，この大きなクラウド環境においても並列速度向上を得ていることが分かる．

表 5 に，頂点数 $|V| = 4 \times 10^6$ および枝数 $|E| = 4 \times 10^7$ となるようにランダム生成したグラフに対する強連結成分分解の Fregel プログラムの実行時間を示す．最短路問題と同様に並列速度向上を得ている．ただし，入力サイズが比較的小さいために

表 4　Amazon EMR 上での最短路問題に対する実行時間 (s)，superstep 数，および総メッセージ量

	P=8	P=16	P=24	P=36	P=48
手書き	341.3	152.1	106.3	77.8	75.7
Fregel	3208	2887	999.6	656.1	598.5

手書き:　superstep 数 = 42，総メッセージ量 = 25.6 GB

Fregel:　superstep 数 = 84，総メッセージ量 = 348.8 GB

表 5　Amazon EMR 上での強連結成分分解に対する Fregel プログラムの実行時間 (s)

	P=8	P=16	P=24	P=36	P=48
愚直な正規化	228.178	119.091	88.696	73.062	63.475
gzip フェーズの削除	212.008	109.349	87.622	69.176	63.955
独立フェーズの同時実行	188.420	100.751	77.463	63.062	54.877

速度向上は落ち込んでいる．表 5 には，また，4 の最後に述べた正規化において可能な最適化を施した場合の計算時間も示している．***gzip*** フェーズの除去により，無駄な組化の処理が消えたため，多少の速度向上が見られるようになった．また，独立したフェーズ `gf` と `gb` の同時計算を行うことにより，SS 数が 55 → 30 と減少し，確かに計算時間の短縮も認められた．

7　おわりに

我々は，大規模グラフ処理のための並列プログラムの作成を支援する新しい関数型領域特化言語 Fregel を提案した．本稿では，その設計から性能評価までを簡単に報告した．我々の目標は，Pig [5] や Hive [10] が MapReduce プログラミングを支援する生産性の高い「言語」を提供したように，グラフ並列計算のための Pregel プログラミングを支援する高レベルな言語を構築することである．

Fregel はグラフを操作する高階関数を提供する関数型言語である．これらの関数を組み合せることで，コンポジショナルにグラフ処理のための並列プログラムを記述できる．この点は，単一の頂点計算関数に全てを単一レベルで記述しなければならない Pregel プログラミングとは大きく異る．プロトタイプコンパイラの生成するコードは，多くの最適化の余地があるにも関わらず，将来性を期待できる性能を示した：良い速度向上を示すとともに，手書きのコードに比べ，幾つかの例では数倍程度の速度低下，最悪の例でも高々 10 倍の遅さにとどまっている．

現在の Fregel コードの絶対性能はまだ満足できるものではない．より詳細な性能解析を行うとともに，その結果を踏まえた最適化を導入したコンパイラの作成が望まれる．また，Fregel モデルの計算のよりフォーマルな定式化や，それに基づく強力な最適化規則の構築も今後の研究のひとつの大きな方向性である．

謝辞 本研究の一部は，JSPS 科研費 JP26280020, JP15K15974 の助成を受けたものです．

参 考 文 献

[1] Apache Giraph: The Giraph Website. `http://giraph.apache.org/`.

[2] Bae, S.-H. and Howe, B.: GossipMap: A Distributed Community Detection Algorithm for Billion-edge Directed Graphs, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM, 2015, pp. 27:1–27:12.

[3] Bahmani, B., Kumar, R., and Vassilvitskii, S.: Densest Subgraph in Streaming and MapReduce, *Proceedings of the VLDB Endowment*, Vol. 5, No. 5(2012), pp. 454–465.

[4] Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A., and Iwasaki, H.: Think Like a Vertex, Behave Like a Function! A Functional DSL for Vertex-Centric Big Graph Processing, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP '16, ACM, 2016, pp. to appear.

[5] Gates, A. F., Natkovich, O., Chopra, S., Ka-

math, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U.: Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience, *Proceedings of the VLDB Endowment*, Vol. 2, No. 2(2009), pp. 1414–1425.

[6] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C.: PowerGraph: Distributed Graph-parallel Computation on Natural Graphs, *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, USENIX Association, 2012, pp. 17–30.

[7] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: A System for Large-scale Graph Processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, ACM, 2010, pp. 135–146.

[8] Salihoglu, S. and Widom, J.: GPS: A Graph Processing System, *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, ACM, 2013, pp. 22:1–22:12.

[9] Sengupta, D., Song, S. L., Agarwal, K., and Schwan, K.: GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM, 2015, pp. 28:1–28:12.

[10] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R.: Hive: A Warehousing Solution over a Map-Reduce Framework, *Proceedings of the VLDB Endowment*, Vol. 2, No. 2(2009), pp. 1626–1629.

[11] Valiant, L. G.: A Bridging Model for Parallel Computation, *Communications of the ACM*, Vol. 33, No. 8(1990), pp. 103–111.

[12] Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., and Bu, Y.: Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees, *Proceedings of the VLDB Endowment*, Vol. 7, No. 14(2014), pp. 1821–1832.