大域的グラフ計算記述言語の 最適化および辺集合に関する拡張

福島 央章[1,a)] 江本 健斗[1]

概要：大規模グラフデータを解析するフレームワークとして頂点主体計算モデルが多数提案されてきた．しかし，頂点集合を扱うようなグローバル視点のアルゴリズムの実装が煩雑になるという問題点がある．その解決のため，頂点集合を用いてアルゴリズムを記述できる領域特化言語 G2PL が提案された．現状，G2PL は辺集合や重み付きグラフなどを用いたアルゴリズムを記述できない．また，頂点主体計算で直接記述したプログラムよりも遅いといった欠点が存在する．本研究では，G2PL に辺集合等の言語機能の拡張と，頂点間メッセージの削減などによる最適化とを追加することにより，問題の解決を試みる．

キーワード：並列プログラミング, プログラミング言語, グラフ計算

Edge-subsets and Optimizations for Global-view Graph Programming Language

Hiroaki Fukushima[1,a)] Kento Emoto[1]

Abstract: The vertex-centric framework has been proposed to reduce the burden of parallel programming for large-scale graph processing. However, it cannot be a great help for implementing global-view graph algorithms described with vertex- and/or edge-subsets, because we cannot use such subsets directly in the framework due to its local-view programming style. G2PL, a DSL for global-view style graph programming, has been proposed to resolve the situation. Users can use vertex-subset variables in their programs to implement such global-view algorithms directly. This study improves G2PL by adding (1) support for edge-subsets, and (2) optimizations to reduce the amount of messages.

Keywords: Parallel Programming, Programming Language, Graph Computation

1. はじめに

SNS のユーザ同士の関係や，Web ページのリンク関係など，グラフ構造を持った大規模なデータが多く存在する．そして，それらを解析することで様々な情報を得ることができる．SNS や WEB グラフなどの実世界に存在するグラフデータは膨大な頂点数であるため，その効率的な解析には大規模グラフを並列に処理する必要がある．そのような背景の中，大規模グラフデータを解析するフレームワークとして Pregel [1] などの頂点主体のグラフ並列計算モデルが多数提案されてきた．

頂点主体計算では，スーパーステップ（以下 SS）と呼ばれる「全頂点が並列に，自身のデータに対する処理と他頂点へのメッセージ送信を行う」という計算ステップを繰り返すことによって，大規模グラフを頂点単位で並列に処理する．頂点主体計算フレームワークを用いる場合，ユーザは，各頂点がひとつの SS 内で実行する逐次処理（頂点計算関数）を記述するだけでグラフ計算の並列実装を実現できる．

しかし，この頂点主体計算は頂点内の処理というローカルな視点でのアルゴリズムの記述を行うが故に，頂点集合

[1] 九州工業大学
Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan
[a)] fukushima.hiroaki197@mail.kyutech.jp

© 1959 Information Processing Society of Japan

を使うアルゴリズムなど，グラフ全体でのグローバルな視点でのアルゴリズムを直感的に記述することができない．その問題に対し，頂点集合を用いてグローバルな視点でアルゴリズムを記述できる領域特化言語 G2PL [2] が提案された．

G2PL は現状，高速な頂点主体計算フレームワークである Blogel [3] のソースコードに変換されるコンパイラが実装されている．ユーザは，頂点集合やその集合演算などを用いたアルゴリズムを G2PL で自然に記述するだけで，頂点主体計算フレームワークで動作する並列プログラムを得ることが出来る．しかしながら，現状の G2PL には以下の 2 点の問題が存在する．

- 辺集合や辺の重みなどを利用したアルゴリズムを記述できない
- 現状のコンパイラでコンパイルされたプログラムの実行速度が，Blogel コードを直接手書きしたプログラムの実行速度に劣る

本研究は，上記の問題の解決のため，G2PL への辺集合等の言語機能の拡張と，コンパイラでの最適化を提案する．

本研究の主な貢献は次のとおりである．

- 辺集合や辺の持つ値を用いたアルゴリズムの頂点主体計算モデルへの対応付け及び変換方法の提案
- G2PL へ追加する辺集合等の言語機能の提案と，既存コンパイラへの実装
- 以下の最適化手法の提案と既存コンパイラへの実装
  - プログラムの順序入れ替えによる SS の削減
  - 辺集合機能により発生する不要な通信の削減
  - 通信データサイズの削減

本論文の構成は次のとおりである．第 2 節で本研究の前提となる頂点主体計算と G2PL を導入する．本研究の提案する辺集合機能とそのコンパイル方法を第 3 節で述べ，同じく提案する最適化手法を第 4 節で述べる．第 5 節で提案手法についての実験結果を示し，最後に第 6 節でまとめを行う．

2. 準備

本節では，本研究の基礎となる Pregel（頂点主体計算モデル）とその実装の Blogel，グローバルな視点でグラフ計算を記述可能な G2PL とそのコンパイル方法について説明する．

2.1 Pregel と Blogel

Pregel [1] は Google が提唱した頂点主体のグラフ並列計算モデルである．Bulk Synchronous Parallel（BSP）モデルに基づき，頂点主体に並列計算を行うモデルである．グラフ内のすべての頂点は BSP モデルに従い，頂点毎の独立した逐次計算（頂点計算）の全頂点に対する並列実行とバリア同期とからなる，スーパーステップ（以下 SS）を繰り返すことでグラフ計算が進行する．

頂点間ではメッセージの送受信によって値のやり取りが行われる．ある SS において他頂点に送信された値は次の SS で利用することができる．

グラフ全体の頂点数などといったグラフ全体の値は，Aggregator と呼ばれる機構によりすべての頂点に対し値の集約を行うことで利用できる．Aggregator もメッセージと同じく集約結果は次の SS で利用可能となる．

各頂点は active，inactive の二つの状態のどちらかをとり，全頂点が inactive になることにより全体の計算が終了する．各 SS で active な頂点は頂点計算を行い，inactive な頂点は行わない．また，inactive な頂点は，他頂点からのメッセージを受信することで avtive に遷移し，次の SS で処理を行う．

ユーザは各頂点が実行する頂点計算関数を記述するだけで並列計算を実現できる．

Blogel [3] は，Pregel モデルの C++ による実装の一つである． 最近のサーベイ研究 [4] により，最も効率よく動作する Pregel 実装として知られている．

2.2 G2PL：頂点集合による並列グラフ計算記述言語

本節では，頂点集合を用いてグローバルな視点でアルゴリズムを記述できる G2PL [2] について説明する．Pregel モデルは大規模グラフ計算用の並列プログラムを安全かつ容易に実装しやすい一方で，頂点単位のローカルな処理記述のため，頂点集合を用いるようなグローバルな視点でのアルゴリズムを直感的に記述できないという問題点がある．その問題点を解決するため，G2PL が提案された． G2PL は頂点集合変数を扱うことのできる手続き型言語であり，他頂点からの集約や if 文・while 文といった制御構造を持つ．G2PL は現状，Blogel ソースコードへ変換されるコンパイラが実装されている．

2.2.1 G2PL による記述

図 1 に最密部分グラフ抽出アルゴリズム（以下 DS アルゴリズム）の G2PL での記述例を示す．以下，このコードに沿って G2PL でのプログラム記述を簡単に説明する．1 行目の `ds()` からプログラム本体の定義が始まる．その内部では，まず，2 行目で頂点集合変数 `S`，`R` を，入力グラフ全体の頂点集合を表す予約語 `V` で初期化している．続く while 文で，`S` が空集合になるまでその内側の処理が繰り返される．繰り返しの条件にある `empty` は空集合を表す定数である．while 文の内側では，まず，頂点集合 `S` の辺密度 `f(S)` からしきい値 `theta` を求めている．この関数 `f` はプログラムの後方 11 行目で定義されている．続いて 5 行目で，頂点集合 `A` として「頂点集合 `S` 内の，次数がしきい値以下である頂点の集合」を求めている．この集合は，右辺の `{}` による内包表記で定義されている．後述するように，この言語での各変数は「全頂点が同じ型の値を

```
1  ds(){
2    S=V; R=V;
3    while(S != empty){
4      theta = 2.0 * (1.0 + 0.01) * f(S);
5      A = {deg <= theta | deg <- Degs(S)};
6      S = S \ A ;
7      if (f(S) > f(R)) then {R = S;}
8    }
9    return R;
10 }
11 f(X) = numEs(X) / 2.0 /numVs(X);
12 numVs(X) = sum [ 1 | u <- V, u in X ];
13 Degs(X) = {sum [ 1 | u <-nvals v, u in X] | v<-V};
14 numEs(X) = sum [ deg | deg <- Degs(X), deg in X];
```

図 1 G2PL によるアルゴリズムの記述例

持つグラフ」に対応し，頂点集合は「各頂点が真偽値をもったグラフ」に対応する．よって，右辺の内包表記は「関数 Degs によって各頂点に対して計算された次数 deg をしきい値 theta と比較して，その頂点上に結果の真偽値（deg <= theta）を置く」という計算により，先に述べた意図通りの頂点集合 A を作っている．その後，6 行目では S から A を引いた差集合を求めて S を更新し，次の行の if 文により頂点集合 S の辺密度が 頂点集合 R の辺密度より大きいならばその S を R に保存して次の繰り返しへ進む．

2.2.2 G2PL での集約

図 1 の 12–13 行目の numVs 関数と Degs 関数の定義を例に，G2PL での他頂点からの集約を説明する．ここで，numVs は，頂点集合 X を受け取り，その集合内の頂点の数を返す関数である．また，Degs は，頂点集合 X を受け取り，入力の頂点集合内での各頂点の次数（X に属する頂点との接続辺の数）を返す関数である．

numVs のように，頂点集合全体からの集約は，[] での内包表記に集約演算子（この例では sum）を付けて表現する．この内包表記は，u <- V で全頂点の集合 V から頂点 u を取り出し，u in X で X に含まれる頂点のみを考慮するようにし，そのような頂点に対して 1 を和に入れることによって「頂点集合 X 内の頂点の数」を計算している．

Degs は各頂点が自身の隣接点に対して集約を行うため，全体としては {} による内包表記を用いて「各頂点が異なる値を持つ」ことを構文的に明示している．その内側では，予約語 nvals を用いた内包表記と集約演算子により，各頂点 v の隣接頂点 u からの値を（v 上に）集約している．

2.3 G2PL から Blogel へのコンパイル

本節では G2PL の Blogel への変換方法について述べる．

2.3.1 G2PL から頂点主体計算モデルへの対応付け

G2PL の頂点主体計算へのコンパイルでは，G2PL の扱う頂点集合というグローバルな情報を，頂点主体計算で各頂点が持つ「その頂点集合に属するか否かの真偽値」というローカルな情報に対応付ける．例として，頂点 a，b，c，d，e からなるグラフの頂点集合 S（= {a,b,c}）についてのイメージを図 2 に示す．頂点 S に属する a，b，c はその頂点の値を T（true）とし，属さない頂点は F（false）とする．これにより頂点集合同士の演算は，和集合が各頂点上の真偽値の論理和に，積集合が論理積になど，各頂点上での真偽値の論理演算に対応付けられる．

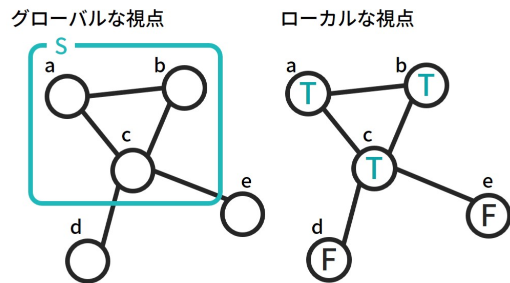

図 2 頂点集合と真偽値の対応付け

プログラム中に現れる値には，DS アルゴリズムにおける theta のようにグラフ全体で同じ値となるものもあるが，G2PL ではこれも「各頂点が同じ値をもったグラフ」として扱う．

2.3.2 集約のコンパイル

G2PL ではグラフ全体からの集約と隣接頂点のみからの集約を統一的に記述できる．しかし，Blogel はこれらを区別するため，前者の集約を「Aggregator による集約」へ，後者の集約を「隣接頂点へのメッセージ送信と，受信したメッセージを結合する Combiner による集約」へと変換する．

2.3.2.1 グラフ全体からの集約

グラフ全体からの集約から Aggregator での集約への変換を numVs(S)= sum[1 | v <- V, v in S] を例に説明する．まず，集約の前の Aggregetor の初期化用に，sum 演算であれば 0 など，集約演算子の単位元で初期化する初期化関数を生成する．そして，各頂点に対する集約処理は，v in S などの内包表記の述語部分の判定を if 文で行い，Aggregator の集約変数に指定の演算子で値を投入するよう生成する：

```
1 virtual void stepPartial(BlogelVertex* v){
2   if (v->value().S) { agg.numVsS += 1; }
3 }
```

2.3.2.2 隣接頂点からの集約

隣接頂点からの集約に対しては，集約に使われる値を隣接頂点にメッセージとして送信する broadcast 関数と，受信したメッセージを集約する combine 関数とが生成され，それぞれを頂点計算関数と Combiner で使用するコードが生成される．例として Degs(S)= { sum [1 | u <- nvals v, u in S] | v <-V } に対して生成されるそれぞれの関数を以下に示す：

```
1 void broadcast(){
```

```
2    if(value().S){
3      vector<VertexID> & nbs=value().edges;
4      for(int i=0; i<nbs.size(); i++){
5        Msg msg;
6        msg.DegsS = value().S ? 1 : 0;
7        send_message(nbs[i], msg);
8      }
9    }
10 }
```

```
1  virtual void combine(Message & old, const
       Message & new_msg){
2    old.DegsS += new_msg.DegsS;
3  }
```

2.3.3 頂点計算関数での状態管理

頂点主体計算では，送信されたメッセージや Aggregator による集約の結果は次の SS でしか利用することができない．そのため，G2PL のコンパイルにあたっては，それらにコンパイルされる集約毎に SS を分割する必要がある．よって，「G2PL プログラム中の実行位置を状態として管理し，各 SS でその状態が指定する実行部分にスイッチする」という動きの頂点計算関数を生成する．

プログラム全体の分割のイメージを図 3 に示す．頂点の値に「実行するステップ（step）」を保持することにより状態を管理し，switch 文により実行部分の選択を行う．黄色の (A) の部分と緑の (B) の部分の間には集約があるため，プログラムはここで分割されてそれぞれ `case 1` と `case 2` になる．また，内部に集約をもつ while 文は，その内側（黄色と緑）と外側（青と赤）とに分割される．この際，while 文の条件は，次に実行する状態として内側（`case 1`）を選ぶか外側（`case 3`）を選ぶかの判断に使われる．

2.3.4 スーパーステップの削減による高速化

前節では集約すべてに対し SS の分割を行うと述べたが，実際の実装では以下の 2 つの条件のいずれかを満たす集約に対しては SS の分割を行わない．

- 既に一度行われている集約で，前回の集約から集約で使われるすべての値が変化していない
- 分割を行ったとき 1 つ前に実行される SS で集約される値が変化しない

前者は以前求めた値と集約結果が同じであるため集約を行う必要がなく，後者は前の SS の集約と同時に集約を行えるためである．

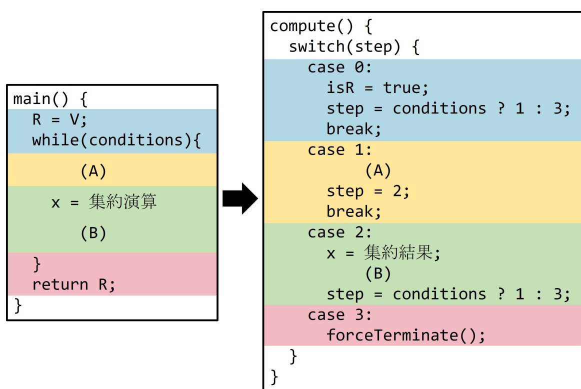

図 3　プログラム分割のイメージ

2.3.5 既存の G2PL の問題点

G2PL は，頂点集合やその集合演算などを用いるアルゴリズムを自然に記述できる一方で，辺についての扱いが弱く，辺集合や辺の重みなどを利用したアルゴリズムを記述できないという問題点をもつ．また，コンパイルされたプログラムの実行速度が，コードを直接手書きした Blogel プログラムの実行速度に劣るという問題点もある．

本研究では，G2PL へ辺集合や辺の持つ値を扱えるようにする機能の拡張を行い，辺の扱いに関する問題点の解決を図る．また，実行速度においても，現状のコンパイラで生成するコードの速度的なボトルネックを解消するよう，その原因であるメッセージ・Aggregator による通信時間やスーパーステップ数を削減する最適化を提案・実装する．

3. 辺集合機能の追加による言語拡張

本節では，前述した辺集合や辺の持つ値などを利用したアルゴリズムを記述できない問題を解決する方法として，G2PL への辺集合に関する機能の拡張を提案する．また，提案した機能について，頂点主体計算モデル上での計算に対応づけるアイデアを議論し，具体的なコンパイル方法について述べる．そのほか，辺集合機能を取り入れるにあたり新たに追加した言語機能などを紹介する．

3.1 辺集合機能を用いたプログラムの記述

まず，具体例として，最大マッチングを求める貪欲近似アルゴリズム [5] の提案機能を用いた記述を図 4 に示す．以下がアルゴリズムの概要である．

(1) 結果のマッチングを表す辺集合変数 `M` を空の辺集合で初期化
(2) 未だマッチしていない頂点の集合 `NotMatched` と，ループ内で新たにマッチした頂点の集合 `Matched` とを，すべての頂点を含む集合で初期化
(3) 以下の処理を `NotMatched` と `Matched` のいずれかが空になるまで繰り返す
  (a) 隣接頂点の中で一番次数の小さい頂点をマッチの候補として，各頂点で求める
  (b) お互いのマッチ候補が一致しているものを確定したマッチとし，それらを辺集合 `A` として定義
  (c) 新たに求められたマッチ `A` を結果の `M` に追加
  (d) 新たにマッチした頂点 `Matched` を `NotMatched` から取り除く

このプログラムは，辺集合変数として，最終的なマッチの集合である `M` とループ内の新たなマッチを表す `A` の 2 つ

© 1959 Information Processing Society of Japan

```
1   mm() {
2     M = emptyE;
3     NotMatched = V; Matched = V;
4     while (NotMatched != empty && Matched != empty){
5       Candidate = argminDegs(NotMatched);
6       A = {id[u] == Candidate[v] && Candidate[u] == id[v] | (u,v) <-Edges(X)};
7       M = M || A;
8       Matched = union[ [u, v] | (u,v) <- A];
9       NotMatched = NotMatched \ Matched;
10    }
11    return M;
12  }
13  Edges(X) = { u in X && v in X | (u,v) <- E };
14  Degs(X) = {sum [ 1 | u <- nvals v, u in X ]| v <- V};
15  argminDegs(X) = {argmin [ u| u <- nvals v, u in X]| v <-Degs(X)};
```

図 4 辺集合機能を用いたプログラムの記述例（最大マッチングの貪欲近似解法）

を使っている．2 行目で，M は，空の辺集合を表す予約語である emptyE で初期化されている．6 行目では，頂点集合と同様に内包表記によって辺集合を定義して，A に代入している．この内包表記では (u,v)<-E で各辺の両端の頂点である u，v を取り出し，u と v のそれぞれの頂点における Candidate 変数の値を Candidate[v] や Candidate[u] という記法で取り出している．一般に，変数 Var の頂点 u における値を（内包表記内の）Var[u] という記法でアクセスできるように G2PL の文法拡張を行っている．なお id は各頂点の頂点 ID として定義された予約語である．7 行目では，頂点集合と同様に，辺集合同士の集合演算を行っている．8 行目では，新たに追加された「和集合をとる集約演算子 union 」を用いることで，辺集合 A の各辺の両端の頂点を集めた頂点集合を定義している．また，13 行目で定義されている Edges(X) という関数では，新たに追加された「頂点が頂点集合に含まれるかを求める演算子 in」を用いて，引数の頂点集合 X に両端の頂点が含まれる辺を集めた辺集合を求めている．

3.2 辺集合機能の頂点主体計算への対応付け

G2PL の頂点集合は頂点が持つ真偽値に対応付けられたが，辺集合もイメージとしては同様に，各辺がその辺集合に属するか否かの真偽値に対応付けることとする．

しかし，辺集合の場合，頂点主体計算への変換では単純に辺の真偽値へと対応付けることはできない．なぜならば，頂点主体計算モデルでは，辺が主体となって値を保持したり計算をしたりすることができないためである．

したがって，辺集合等に関連する「辺の値と計算」の頂点主体計算モデルへの変換には，辺に関する情報をどのように頂点上に持たせて，その関連する計算をどのように各頂点上の計算に対応付けるかを考えなければならない．両端の頂点を a, b，辺の持つ値を e_{ab} としたとき，両端の頂点 a, b が e_{ab} を重複して持つ手法や，両端の頂点 a, b のうちどちらか一方が e_{ab} を持つ手法などが考えられる．本研究では，比較的単純でメッセージ送信や SS 数で無駄の少ない前者の手法を採用する．

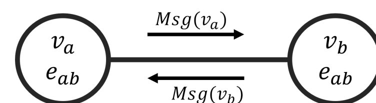

図 5 両端の頂点が辺の情報を持つ手法のイメージ

本研究での「辺の値と計算」の扱いのイメージを図 5 に示す．両端の頂点 a, b が辺の持つ値 e_{ab} を重複して持ち，辺の値に対して計算を行うときには，両端の頂点が同時に重複して同じ計算を行う．また，辺の計算で両端の頂点の値を利用する場合には，事前にお互いの頂点がメッセージとして頂点の値を送りあっておく．

3.3 提案する辺集合と関連機能

本研究で新たに G2PL に追加した機能を以下にまとめる．

- 辺集合変数と集合演算
- 辺の重みなどの辺が持つ値
- 辺集合に関する定数：入力グラフのすべての辺の集合を表す E と空の辺集合 emptyE
- 内包表記による辺集合の定義
  (e.g., `A = { Weight[e] < x | e <- E };`)
- 辺集合での集約
  (e.g., `A = sum[ 1 | (u, v)<- E ];`)
- 内包表記内からの頂点の値へのアクセス
  (e.g., `A = { Val[u] + Val[v] | (u, v)<- E };`)
- 集約演算子 union，argmin，argmax

以下，非自明な機能についてその実現方法を補足する．

3.3.1 辺集合変数と内包表記による辺集合の定義

辺集合や辺の持つ値を両端の頂点に重複して持たせることから，各頂点は，辺集合変数一つにつきその頂点の次数分の真偽値を持たねばならない．したがって，コンパイルされた Blogel コードでは，VertexID を頂点 ID の型であるとして，辺集合変数 A が map<VertexID，bool> 型の値

に変換される．辺集合変数に関する計算の変換は頂点集合変数に対するものとほぼ同じであり，計算を map の全要素に対して行う点だけ異なる．

3.3.2 辺の持つ値とその集約

辺の持つ値も辺集合と同様に，頂点に 頂点 ID をキーとした map に変換される．また，辺集合についての集約は，G2PL の頂点集合の集約の組み合わせに置き換えられて Blogel へと変換される．例えば，a = sum[1 | (u, v)<- E]; は次の集約の組み合わせになる：

```
1  _endpnt_a = {sum[ 1 | u <- nvals v]| v<-V};
2  a = sum[ _endpnt_a | v <- V ];
```

この様な変換をしているのは，辺集合の集約においてその集約される値の計算に両端の頂点の値が必要である場合，すなわち，(u, v)<- A の u と v が使われる場合に，両端頂点の値を送り合う通信が必要だからである．

内包表記の生成子部分（? <- E）については幾つかの記法が用意されている．辺の重みなどの辺の値にのみアクセスするときは sum[Weight[e] | e <- E] のように，両端の頂点の値にのみアクセスするときは sum[Degs[u] + Degs[v] | (u, v)<- E] のように記述できる．これらは，辺の値と両端の頂点の値のどちらか一方を利用する場合の特殊な記述であるが，一般にそのどちらの値も利用する場合は sum[Weight[e] + Degs[u] + Degs[v] | (u, e, v)<- E] のように記述できる．

3.3.3 両端の頂点へのアクセスでの一貫性

辺の両端の頂点による辺の値と計算の重複により，A = { A[u] + A[v] * 2 | (u, v)<- E }; や A = { A[u] | (u, v)<- E }; のような u と v に関して対称でないプログラムに対して，単純に u を隣接頂点，v を自頂点のような変換を行ってしまうと，両端の頂点で計算される辺の値の計算が異なる結果となってしまう．そのため，内包表記で (u, v)<- A として両端の頂点を取り出した場合，頂点 ID が小さいほうが u 側，大きい方が v 側となることとした．これにより，計算の一貫性が保たれる．

4. 通信とスーパーステップの最適化

本節では，既存の G2PL コンパイラでコンパイルされたプログラムの実行速度の問題を解決する方法として，三種類の最適化を提案する．具体的には，プログラムの順序入れ替えによる SS の削減，辺集合機能により発生する不要な通信の削減，通信データサイズの削減の三種類である．

4.1 プログラムの順序入れ替えによる SS の削減

G2PL の Blogel へのコンパイルでは，基本的に集約毎に SS が分割される形のコードが生成される．これに対し，既存のコンパイラは，以下の条件のいずれかを満たしたときについて SS の分割を抑制する最適化を行っている．

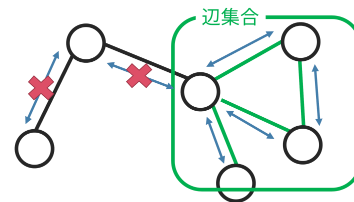

図 6 不要な通信の削減による最適化のイメージ

- 既に一度行われている集約で，前回の集約から集約で使われるすべての値が変化していない
- 分割を行ったとき 1 つ前に実行される SS で集約される値が変化しない

ここで次のようなプログラムを考える．

```
1  a = x;
2  b = sum[ a| v <- V];
3  c = x;
4  d = sum[ c| v <- V];
```

このプログラムでは集約を行っている箇所が b と d の 2 か所存在し，b は a に，d は c にそれぞれ依存している．このプログラムをコンパイルでは，まず，基本の SS 分割によって「1 行目」と「2, 3 行目」と「4 行目」の 3 つの SS への分割が行なわれる．そして，前述の 2 つの条件を確認するとそのどちらも満たされないため，既存のコンパイラはそのまま 3 つの SS として Blogel コードを生成する．

しかし，このプログラムは，実際には 2 つの SS だけで計算することが可能である．まず，2–3 行目の b と c に注目すると，これらは互いに依存していないことから，この 2 つの文を入れ替えてもプログラムの実行結果は変わらないことに気づく．そして，実際にこれらを入れ替えてみると，次のようになる：

```
1  a = x;
2  c = x;
3  b = sum[ a| v <- V];
4  d = sum[ c| v <- V];
```

この入れ替え後のコードは，上述の最適化条件の 2 つ目を満たす．そのため，b と d が同時に集約されるようになり，2 つの SS だけからなるコードにコンパイルされる．

このように文同士に依存関係がなく入れ替えても実行結果が変わらないものを見つけ出し，より SS 数が少なくなる入れ替えを探すことで，SS 数の削減を試みる最適化を行う．

4.2 辺集合機能により発生する不要な通信の削減

A = sum[expr | e <- E, e in S]; のような条件付きの集約では，その条件を満たさない辺についての値の送受信は無駄である．よって，その様な不要な通信の削除を行う．最適化のイメージを図 6 に示す．

例えば，上記のコード片から生成される，最適化なしの

© 1959 Information Processing Society of Japan

愚直な Blogel コードでの通信部分は次のようになる：

```
1 for (const VertexID& nb_id : value().edges) {
2   Message msg;
3   msg._endpnt_A = value().S[nb_id] ? expr : 0;
4   send_message(nb_id, msg);
5 }
```

この最適化を導入することにより，通信部分のコードは以下の形になる：

```
1 for (const VertexID& nb_id : value().edges) {
2   if (value().S[nb_id]) {
3     Message msg;
4     msg._endpnt_A = expr;
5     send_message(nb_id, msg);
6   }
7 }
```

この最適化後のコードでは，辺集合 S に含まれる辺のみについて通信が行なわれており，辺集合のサイズが小さい場合には効果的にメッセージ数を削減できる．

4.3 通信データサイズの削減

本節では，集約等のために送受信される通信データのサイズ縮小による最適化について述べる．

既存の G2PL コンパイラで生成されるコードは，SS 分割後のプログラムが「実行ステップ 1 で msg1 の値を送信し，次の実行ステップ 2 で msg2 を，実行ステップ 3 で msg3 を . . .」といった通信を行うものであった場合，全実行ステップの通信において「msg1，msg2，msg3 . . . の全てを保持したデータ」を送受信する（そのステップで実際に使われない部分には未定義の値が入っている）．そのため，多くの集約を用いて記述された G2PL プログラムからコンパイルされたコードでは，各通信に占める無駄なデータの割合が大きくなり，無駄に大きな通信コストを発生させることになってしまう．

本研究では，ステップごとに必要な値のみを送受信するように改善を行う．具体的には，通信データにビットフラグを追加し，どのデータを通信データ内に含んでいるのかを一緒に送受信する手法を導入する．

通信データサイズの削減による最適化のイメージを図 7 に示す．ステップ 1 では msg1 と msg3 が，ステップ 2 では msg4 のみの通信が必要であるとする．flag は，例えば i ビット目を msg$i$ に対応付けるとすると，ステップ 1 では 1 ビット目と 3 ビット目が立った 000101b に，ステップ 2 では 001000b に設定する．そして，実際のデータ送信時に，立っているフラグに対応するもののみをバイト配列へまとめるシリアライズを行う．

5. 実験と評価

本研究で実装したコンパイラによってコンパイルされたプログラムに対する性能評価を本節で述べる．この評価には図 4 の最大マッチング計算のプログラムを用いた．

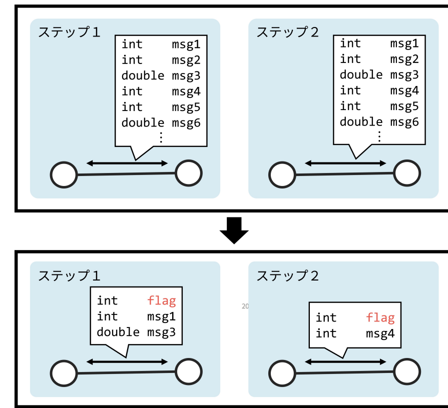

図 7　通信データサイズの削減による最適化のイメージ

表 1　入力グラフの一覧

	頂点数（$\|V\|$）	辺数（$\|E\|$）
random 1M	$1.0 * 10^6$	$1.0 * 10^7$
random 2M	$2.0 * 10^6$	$2.0 * 10^7$
random 4M	$4.0 * 10^6$	$4.0 * 10^7$
wikipedia 1M	$1.0 * 10^6$	$1.0 * 10^7$
wikipedia 2M	$2.0 * 10^6$	$2.0 * 10^7$
wikipedia 4M	$4.0 * 10^6$	$4.0 * 10^7$

表 2　実行環境

OS	Ubuntu 18.04.5 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	1.2.1

5.1 実験環境と使用した入力グラフ

本実験で使用した入力グラフは表 1 に示した 6 種類である．前者 3 つのグラフは Watts-Strogatz モデルに基づいて，繋ぎ変え確率を 0.5 として作成したランダムグラフである．また，後者 3 つのグラフは，現実に存在するグラフとして Wikipedia から取り出した Web グラフである．ただし，どのグラフも無向グラフとして扱う．

実験で使用した実行環境は表 2 に示す PC16 台からなるクラスタである．

5.2 並列実行による速度向上の比較

本節では，本研究で提案した辺集合機能の並列実行による速度向上を確認する．入力グラフに頂点数 1M のランダムグラフを使用し，ワーカ台数を変化させて測定した通信時間，SS 計算等の時間，合計実行時間を，最適化の有無でそれぞれ表 3 に示す．

「最適化なし」では，SS の計算は比較的，台数が増えるにつれ実行時間が短くなっていることが確認できる．しかし通信時間や合計実行時間では，並列実行による速度向上

© 1959 Information Processing Society of Japan

表 3　最大マッチング計算の並列実行による実行時間（s）

	最適化あり			最適化なし		
	通信時間	SS 計算時間等	合計時間	通信時間	SS 計算時間等	合計時間
P=1	11.245	38.719	49.964	33.229	91.912	125.142
P=2	33.81	19.561	53.371	134.085	43.339	177.424
P=4	22.459	7.4	29.859	81.788	17.07	98.858
P=8	19.558	3.683	23.241	67.741	7.005	74.746
P=12	17.944	2.616	20.56	62.781	4.837	67.617
P=16	17.179	2.278	19.457	60.92	3.774	64.693

表 4　ワーカ 16 台，最大マッチング計算での SS 数，メッセージ数，実行時間（s）

	最適化あり			最適化なし		
	SS 数	メッセージ数	時間	SS 数	メッセージ数	時間
random 1M	112	457267206	19.457	131	1300000000	64.693
random 2M	112	914497624	39.295	131	2600000000	130.997
random 4M	112	1829124702	81.248	131	5200000000	264.600
wikipedia 1M	40	192904261	5.429	47	460000000	17.104
wikipedia 2M	46	430541571	11.897	54	1060000000	32.094
wikipedia 4M	58	1083349593	24.665	68	2680000000	69.504

は芳しくないことが確認できる．特に 1 台から複数台に増やしたときの実行時間の増加が大きい．これは新たに追加した，辺集合機能のメッセージの送信数が多く計算時間の全体のうち，メッセージの送受信の時間の割合が高いことが原因であると考えられる．

「最適化あり」では，「最適化なし」と比較して全台数で実行時間が短くなっており，通信時間においても改善が確認できる．台数効果による速度向上は芳しくないが，台数が増えていくにつれ合計実行時間の中の通信に使われる時間の割合が増えているからであり，12 台や 16 台では，合計実行時間の 9 割弱が通信による時間であることが確認できる．また，通信の削減により，単一ワーカから 2 台へ並列数を変化させたときの速度低下が小さくなっていることが確認できる．

5.3　入力グラフ毎の実行時間の比較

ワーカ台数 16 台での 6 つのグラフをそれぞれ入力として計測した SS 数，メッセージ数，実行時間を表 4 に示す．

最適化の導入により，SS 数とメッセージ数のどちらも減少していることが確認できる．また，実行速度で比較すると，2.6 倍から 3.4 倍程度の速度向上が確認できる．

6.　おわりに

本研究では，G2PL の記述性向上の為，G2PL へ追加する辺集合等の言語機能の提案と，既存コンパイラへの実装を行った．また，既存のコンパイラの問題であった実行速度の改善に向け，スーパーステップと通信を削減する最適化を行い，一定の実行時間の向上を実現した．

今後の課題として以下が考えられる．今回の辺集合機能の導入により，プログラム内に頂点集合と辺集合が混在するようになった．これにより，プログラムの可読性低下の問題が生じてしまっている．ユーザがこれらの変数を区別できるように，必要に応じて型アノテーションを付記できるような文法の導入が望まれる．また，より多くのアプリケーションを記述できるように，乱択や平均値計算などの新たな集約演算子の導入も望まれる．さらに，通信コストのさらなる削減のための最適化の導入や，効率的な部分グラフ主体計算へのコンパイル手法の開発なども課題として挙げられる．

謝辞　本研究は JSPS 科研費 JP19K11901, JP19K11903 の助成を受けたものです．

参考文献

[1] 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*, pp. 135–146 (2010).

[2] Emoto, K. and Sadahira, F.: A DSL for graph parallel programming with vertex subsets, *Journal of Supercomputing*, Vol. 76, No. 7, pp. 4998–5015 (2020).

[3] Yan, D., Cheng, J., Lu, Y. and Ng, W.: Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs, *Proceedings of the VLDB Endowment*, Vol. 7, No. 14, pp. 1981–1992 (2014).

[4] Ammar, K. and Özsu, M. T.: Experimental Analysis of Distributed Graph Systems, *Proceedings of the VLDB Endowment*, Vol. 11, No. 10, pp. 1151–1164 (2018).

[5] Lim, B. and Chung, Y. D.: A Parallel Maximal Matching Algorithm for Large Graphs Using Pregel, *IEICE Transactions on Information and Systems*, Vol. E97.D, No. 7, pp. 1910–1913 (2014).

© 1959 Information Processing Society of Japan