頂点主体並列グラフ処理の制約解消器による効率化

森畑 明昌　江本 健斗　松崎 公紀　胡 振江　岩崎 英哉

本発表では、分散並列処理における無駄な通信や同期の除去などの効率化を制約解消器を用いて自動的に行う方法について、頂点主体並列グラフ処理を題材に報告する。頂点主体並列グラフ処理とは、各頂点が行う処理を指定することで大規模グラフを分散並列処理する枠組みである。我々が以前提案した関数型言語 Fregel では、各頂点が行う計算を宣言的に記述し通信や同期などは明示しないため、無駄な計算・通信・同期などが発生しており、また頂点主体グラフ処理特有の効率化も行えていなかった。これに対し、効率化が可能な条件を論理的に記述し、それを制約解消器で確認することで効率化を行う手法を提案する。これにより、効率化がデッドロック等のバグをもたらさないことも自動的に保証される。本発表では、限定記号除去器を利用する方法と SMT ソルバを利用する方法を比較しながら提案手法を説明し、予備的な実験結果について報告する。

1　はじめに

近年では、インターネットを初めとする情報技術の発展により、一台の計算機ではとうてい処理できないような大規模なデータも珍しくなくなってきている。このようなデータを扱うには分散並列処理が必須となるが、一般に効率の良い分散並列プログラムを構築するのは容易でない。分散並列処理には通信・同期・負荷分散など逐次処理では現れない困難が多く現れる。これらを不適切に用いれば、デッドロックや正しくない結果が非決定的に現れるなど、除去の難しいバグを埋め込んでしまうことになりかねない。細かく通信や同期を繰り返せばこのようなバグを避けることができるが、その場合、逐次処理に比べかなりのオーバーヘッドが生じ、また負荷分散も難しくなりがちであり、良い実行速度が得られないことが多い。

性能の良い分散並列処理を比較的簡単に達成するための一つのアプローチは、プログラマは性能を気にせず正しいプログラムを記述し、コンパイラが注意深い解析によって効率の良いコードを生成する、というものである。このアプローチは宣言的並列処理言語などの文脈で長らく研究されてきた。本発表では、特に頂点主体並列処理 [13] を題材に、このアプローチの可能性について調査した結果を報告する。

頂点主体並列処理は、大規模グラフ処理を分散並列実行するためのフレームワークである。ソーシャルネットワークや顧客・商品関係ネットワーク、タンパク質反応ネットワークなど、近年では大規模なグラフが多数手に入るようになり、その処理・解析を分散並列実行したいという要求が高い。しかし、グラフは配列や木構造の場合とは異なり、自明なデータの分割方法がなく、またグラフ処理はグラフ全体を繰り返し走査するようなアルゴリズムが多いため独立なタスクに分割しづらく、分散並列処理を実装することが難しくなりやすい。頂点主体並列処理では、「各頂点が毎ステップどのような計算を行うか」という観点でグラ

Akimasa Morihata, 東京大学, The University of Tokyo.
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.
Hideya Iwasaki, 電気通信大学, The University of Electro-Communications.

* Optimizing Vertex-centric Parallel Graph Processing using Constraint Solver

```
sssp g = let init v = if v が始点 then 0 else ∞
             step v prev = min (prev v) (minimum [ prev u + e | (e, u) <- is v ])
         in fregel init step Fix g
```

図 1　単一始点最短路問題に対する Fregel プログラム

フ処理を捉えることにより、この困難を軽減する。

例として、始点から各頂点までの最短路の長さを求める計算（以下 SSSP と呼ぶ）を考える。図 1 が、頂点主体並列処理のための関数型言語 Fregel [5] で記述した SSSP のプログラムである。ただし、読みやすさのため、頂点に対応するデータ構造を省略するなど、幾つか簡略化している。Fregel は Haskell のサブセットであり、その中核はグラフを処理する高階関数 `fregel` である。その第 1 引数は初期化時に各頂点で行う計算を表す。今回の場合、各頂点 `v` に対し、それが始点かどうかを加味して距離の初期値を求めている。第 2 引数は処理の各ステップで各頂点に対し実行される。今回の場合、各ステップにおいて、各頂点 `v` では、直前のステップでの各頂点の計算結果を格納するテーブル `prev` を用いてその頂点までの距離を更新する。新しい距離は、その頂点の前回の結果 `prev v` と、隣接頂点 `u` の前回の結果 `prev u` に接続辺の重み `e` を加えたものの最小値となる。なお `is` は当該頂点に接続している辺の重みとその辺の他端頂点の対のリストを返す。第 3 引数は停止条件であり、今回の場合この計算はどの頂点の値も変化しなくなったら終了する。

頂点主体並列処理ではどのフレームワークでも、記法と細部は違えど似たようなプログラムを記述する。すなわち、各頂点で毎ステップ、周辺の頂点の計算結果（ないしは周辺の頂点から送られてきたメッセージ）を元にどのような計算を行うのか（またどのようなメッセージを送るのか）、を記述する。フレームワークはこれらの記述をもとに分散並列グラフ処理を実行する。やはり細部に違いはあるが、おおむね以下のような処理となる。

- グラフの頂点集合は分割され各計算ノードに分配される。
- 各計算ノードは毎ステップ、自分に分配された頂点の処理を並列に行う。
- 計算ノードを跨ぐ情報は通信によって受け渡す。

これにより、プログラマは分散並列処理の実際の状況、例えば通信や同期など、を意識することなくプログラムを記述することができる。しかも、各頂点では概ね同じ処理を行うため、頂点を適切に計算ノードに分配すれば良い負荷分散が得られる可能性が高い。このように頂点主体並列処理はグラフの並列分散処理の難しさを軽減することができるため、近年注目を集めいくつものフレームワークが提案されている [5–7, 10–12, 20, 25, 28]。

しかし、SSSP のようなかなり簡単な問題でも、実際には効率の良い頂点主体並列処理プログラムを書くのはそれほど簡単ではない。図 1 のプログラムは以下の点で非効率である。

- プログラムの文面通りに読めば、全ての頂点が全てのステップで計算を行うことになる。しかし、実際にはそれは必要ない。隣接頂点の距離が更新された頂点のみが計算をすれば十分である。
- 上と関連する話題だが、プログラムの文面上は（隣接頂点が別の計算ノードにあれば）全ての頂点が毎ステップ通信を行う。しかし、上記の通り、自分の距離が更新された場合にのみ通信を行えば十分である。
- そもそも、このプログラムは Bellman-Ford のアルゴリズムと同等であり、グラフの大きさを n として $O(n^2)$ 程度の計算を要する。しかし、もしダイクストラ法を用いることができれば $O(n \log n)$ 程度の計算で済む。つまり、逐次実装に比べこの分散並列実装は漸近計算量でもかなり遅い。ダイクストラ法は分散並列実装に向かないとしても、この差は無視しがたい。より適切なアルゴリズムの利用が望ましい。

既存の頂点主体並列処理フレームワークでも、これら問題を解決するための機能を提供している場合が多い。例えば、Pregel [13] では、頂点を一時的に inactive

にする機能を提供し、また送るべきメッセージをプログラマに明示的に特定させることで、1 番目と 2 番目の問題への対処を可能としている。また、頂点主体並列処理の亜種である部分グラフ主体並列処理 [20, 25] や、非同期の頂点主体並列処理で可能となる頂点処理順序の優先度スケジューリング [3, 11, 18] を用いることで、部分的にダイクストラ法を取り入れ並列性と逐次性能を両立させることができる（詳細は 2.3 節参照）。しかしいずれにせよ、これら機能を用いるためには明示的なプログラミングが必要である。頂点の inactive 化、通信の削減、部分グラフに対する特別な処理、また優先度の設定などは、いずれも不適切に用いると計算結果が正しくなくなってしまうものであり、その利用は容易でない。

以上の状況を背景に、本発表では、制約解消器を用いることでこれらの効率化を自動的に導入する手法を提案する。提案手法では、図 1 のように効率を気にせず記述した頂点主体並列処理プログラム（具体的には Fregel で記述されたものを想定する）を入力とし、制約解消器を利用したプログラム解析によって、頂点の inactive 化や通信の削減等が可能かどうかを判定する。

制約解消器としては限定記号除去器 [2] と SMT ソルバ [4] の 2 つを考える。限定記号除去は、限定記号（$\forall$ ないし $\exists$）を含む論理式を、それらを含まない等価な論理式に変換する手法である。これは限定記号の任意の深さのネストを扱えるため強力である。さらに、限定記号除去によって得られる式は効率的な実装のために必要なプログラム断片も与える。しかし、限定記号除去は、制約解消に時間がかかる、制約解消が可能な論理体系が少ないなど、実用上は扱いが難しい。これに対し、SMT はいずれか 1 つの限定記号のみを含む論理式を入力とし、その真偽を判定するもので、限定記号除去に比べ表現力は落ちるが、高速なソルバが近年開発されているため実用的な実装が可能である。以上の状況をふまえ、本発表では、まず効率化を限定記号除去問題として定式化し、その現実的な実装として SMT ソルバを用いる手法を提案する。さらに、提案手法の有用性の確認のため、SMT ソルバとして Z3 ソルバ [†1] を用いた予備的実装を行った結果についても報告する。

本発表は Fregel で記述された特定の形式の並列プログラムの効率化を扱っているに過ぎない。しかし、本発表で採り上げている問題は分散並列プログラム記述の際には普遍的に現れるものである。例えば、並列グラフ処理に関しては、頂点主体並列処理以外の枠組みでも、宣言的な記述から高度な効率化技法によって効率的な分散並列プログラムを得る手法は近年いくつも提案されている [3, 18, 19, 22]。また、SMT による実装の際に現れる幾つかのヒューリスティクスを除き、本発表のアプローチは Fregel や頂点主体並列処理に特化したものではない。そのため、本発表の内容は分散並列計算における無駄な通信や同期の削減等についての一般的な知見を与えるものと期待している。

2 頂点主体並列処理

2.1 Pregel

Pregel [13] は Malewicz らによって提案された頂点主体並列処理フレームワークである。以降に現れた頂点主体並列処理フレームワークのほとんどがこれを基にしているため、本稿でもまず Pregel の紹介を行う。

Pregel は、手続き型言語と関数型言語という違いはあるが、Fregel と同様、各頂点が各ステップで実行する処理を指定する。各頂点はローカルな記憶領域をもち、この記憶領域の更新が各ステップでの処理の主な内容である。この処理の記述に際しては以下の特別な機能を用いることができる。

- 受け取ったメッセージの読み出し
- 他の頂点へのメッセージの送信
- 次のステップからその頂点が inactive になるという宣言
- 全ての active な頂点からの値の集約（アグリゲータ）

メッセージの送信は、送信先頂点の識別子が既知であれば任意の頂点に対し可能である。しかし、各頂点では通常グラフの大域的なトポロジーの情報が利用できないため、ほとんどの場合は隣接頂点へのメッ

†1 Z3 Solver: `https://z3.codeplex.com/`

セージの送信となる。

各頂点は active ないしは inactive であり、上述の通り、自らの宣言によって inactive になることができる。他の頂点からのメッセージを受け取ると inactive な頂点は active になる。なお、メッセージを受け取らなかった場合、inactive な頂点での処理はスキップされる。全ての頂点が inactive になりメッセージも送出されなかった場合プログラムは終了する。

大域的な情報（例えば総和、平均など）の取得のためアグリゲータと呼ばれる機能が提供されている。これは、全ての active な頂点から指定された値を集約し、各頂点に伝えるものである。

Pregel のプログラムは分散環境上で実行できる。グラフは頂点単位で各計算ノードへ分配される。なお、辺は頂点に付随するデータとして扱われる。実行は BSP (Bulk Synchronous Parallel) [26] の方式に則っている。各計算ステップは、各頂点の処理、メッセージの送受信、そしてメッセージの待ち合わせを兼ねる大域バリアから成る。そのため、メッセージは次の計算ステップになるまで到着せず、また次の計算ステップになれば必ず到着する。この挙動は、同一計算ステップ内では各頂点の処理が完全に独立であることを保証しており、並列計算を可能としている。このような計算ステップを全頂点が inactive になるまで繰り返すのが Pregel プログラムである。

2.2 Fregel

Fregel [5] は Pregel を元にした関数型言語であり、明示的なメッセージ送受信や inactive 化によるプログラム停止などを隠蔽することで記述性を向上させることを目的としている。Fregel は Haskell の部分集合となっており、基本的な記法の意味は Haskell のそれと同じである。

例として図 1 のプログラムを再訪する。Fregel の核となるのは高階関数 `fregel` である。これは引数としてプログラム停止の条件をとるため、各頂点の処理の内部に停止のためのコードを記述する必要がない。また、通信は「前回の値」を表すテーブル（図 1 の `prev`）の値の参照として暗黙に記述される。テーブル読み出しのキーとしては `is v`（頂点 `v` に向かう辺とその他端を返す）という特殊な関数（以下ジェネレータと呼ぶ）によって得られた `v` の隣接頂点が用いられている。これ以外にも、グラフ中の全ての頂点を返すもの（アグリゲータの実現に利用）など、いくつかのジェネレータが提供されている。これらの機能により、「次に使われるであろう値を想像してメッセージ送る」ことを避け、「今回の計算に必要な値を読み出す」形式でのプログラム記述が可能となっている。

読み出した値はリストとなり、それを適切な演算（図 1 の場合 `minimum`）で集約する。読み出した値は本質的には多重集合であるため、通常のリスト処理関数で扱うのは不適切である。そのため、集約は結合的かつ交換的な演算子（加算、乗算、最大値、最小値など）によるものに限っている。また、メッセージが一つも届かない場合があるため、集約の演算子は単位元をもつことを仮定している。

Fregel の処理系には 2 種類ある。一つは既存の Haskell 処理系での実行を行うもので、主にテストのために用いることを想定している。この処理系は本発表の趣旨と関係ないので以降無視する。もう一つは、既存の頂点主体並列処理フレームワークのプログラムへと翻訳するもので、並列分散実行のために用いる。現時点では Giraph[†2] および Pregel+ [29][†3] のコードを出力できる。Giraph は Java、Pregel+は C++言語であるという違いはあるものの、いずれも機能としては Pregel に非常に近い。

2.3 頂点主体並列処理の亜種と最適化

Pregel で提案された頂点主体並列処理が単純であることもあり、早くからそのモデルの問題が指摘されて、様々な改善が提案されている。ここでは特に Pregel の性能上の問題に関連するものについて述べる。

Pregel の BSP モデルの採用については議論が多い。BSP モデルは大域的な同期に従って計算が決定的に進むため計算の挙動は分かりやすいが、計算ノードが多い場合には大域的同期のコストはかなり大きい。一方、多くのグラフアルゴリズムは非同期的に実行してもよい。ここで「非同期実行」とは、大域的な

†2 Apache Giraph: `http://giraph.apache.org/`
†3 Pregel+: `http://www.cse.cuhk.edu.hk/pregelplus/`

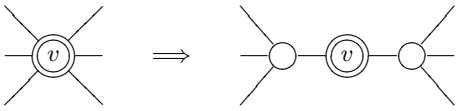

図 2　高次数頂点の分割

同期毎に各頂点を 1 回ずつ走査するのではなく、各頂点を適当なスケジューリングに従って走査し、走査の瞬間に利用可能なメッセージを用いて計算を進めるものである †4。非同期実行を採用している既存フレームワークも少なくない [6,10,12]。また、非同期実行と BSP モデルでの実行を併用することでさらに効率を高める方法も提案されている [11,28]。

非同期実行は同期を取り除くことができるだけでなく、さらなる最適化をもたらしうるという点でも重要である。以下 2 つ例を挙げる。

高次数頂点の分割　現実に現れるグラフの多くでは、非常に次数の高い頂点が少数存在する。各頂点が受け取るメッセージの量は概ね次数に比例すること、往々にして各頂点の仕事のほとんどはメッセージの集約であることもあり、次数の高い頂点がボトルネックになることは多い。このボトルネックを避けるため、図 2 のように次数の高い頂点を複数の頂点に分割する手法がよく用いられる [27,29]†5。分割で得られた各頂点はそれぞれに届くメッセージの集約のみを担当し、最終的にその結果が当該頂点に届いたらその頂点の処理を行う。メッセージの送信も同様である。しかし、頂点分割は BSP モデルでは単純には適用できない。高次数頂点に届く・高次数頂点から送られるメッセージは、本来届くべきであった計算ステップより遅れることになるため、他の頂点との計算の同期が崩れてしまう。よって、BSP モデルでは分割された頂点に対しては追加の特別な処理を行うよう実装を拡張する必要がある。一方、非同期実行を前提としていれば、特定頂点の実行が他の頂点より遅れることは想定の範囲内であるため、特別な取り扱いは不要になる。

優先度実行　グラフ処理では往々にして全ての頂点の処理が同等に有意義であるとは限らない。例えば SSSP では開始点から遠い点の処理はいくら行っても最終結果にほとんど貢献しない。このことをふまえ、非同期実行が可能であれば、開始点から近い点の処理を優先的に行うスケジューリングによってより高速に最終結果を得ることができる。優先度実行の機構を提供しているシステム [3,11,18] も複数ある。

また、「頂点主体」というモデルそのものにも疑問が投げかけられている。頂点という単位はグラフ処理としてはあまりに粒度が細かい。実際には各計算ノードは部分グラフを保持しており、より広い範囲の情報を用いて高速な処理ができる可能性がある。例えば、SSSP であれば、各部分グラフ内ではダイクストラ法を使うというのも合理的である。このような観点から、部分グラフ主体（または近傍主体）の計算モデルも提案されている [20,25]。このモデルでは、ユーザは頂点ではなく部分グラフを処理するアルゴリズムを記述する。

部分グラフ主体並列処理は非同期的実行と似た部分がある。非同期実行の観点からは、部分グラフ主体並列処理は同一計算ノード内にある部分グラフを計算ノードを跨ぐメッセージ送受信に優先して行うスケジューリングと見なすことができる。部分グラフ主体並列処理はこれに加え部分グラフに特化したアルゴリズムを用いることができる点で異なるが、これもスケジューリングの一部と考えられる場合もある。例えば SSSP であれば、優先度実行を行えば部分グラフ内の処理はダイクストラ法に変更できる。ダイクストラ法と Bellman-Ford 法を組み合わせることで同一ノード内での計算の逐次性能と全体としての並列性を両立させるのは、SSSP の高速な並列アルゴリズムである Δ-stepping 法 [14,18,22] などにも繋がりうるものである。

†4　実際には、フレームワークによって非同期実行の実態は異なる。例えば gather-apply-scatter モデル [6,12] では頂点の走査順序は任意だが、その頂点が必要なメッセージは全て手に入っている必要があるというモデルである（実装上はそのために古いメッセージをキャッシュする）。

†5　このアプローチは、頂点を計算ノードに分配するのではなく辺を分配していると考えることもできる。

$$
\begin{aligned}
\texttt{step v prev = let } c_1 &= \textstyle\bigoplus_1 [f_1(\texttt{e}, \texttt{prev u}) \mid \texttt{(e,u) <- is v, } p_1(\texttt{prev u})] \\
&\vdots \\
c_n &= \textstyle\bigoplus_n [f_n(\texttt{e}, \texttt{prev u}) \mid \texttt{(e,u) <- is v, } p_n(\texttt{prev u})] \\
\texttt{in } & g(\texttt{v}, c_1, \ldots, c_m)
\end{aligned}
$$

図 3　提案手法で扱うプログラムの一般形

3　限定記号除去と SMT

3.1　限定記号除去

限定記号除去とは、限定記号（$\forall$ または $\exists$）を含む論理式をそれと等価で限定記号を含まないものへと変換する手法である。例えば、$\forall x.\ x^2 + ax + b > 0$ を $4b - a^2 > 0$ に変換するのが限定記号除去の一例である。特に論理式中の変数が全て限定記号によって束縛されている場合、限定記号除去後の式の真偽は自明であるため、論理式の真偽判定の一手法として用いられる。

命題論理では、真偽値全てを単に列挙すれば良いので、限定記号除去は常に可能である。述語論理の場合には限定記号除去の方法が知られている体系は限られており、Presburger 算術や実閉体などが挙げられる。命題論理の場合の議論からも分かるとおり、限定記号除去が例え可能であったとしても、その計算量は理論上も実用上も非常に大きい場合が多い。詳細は専門書 [2] などを参照されたい。

3.2　SMT

SMT（Satisfiability Modulo Theories）[4] は命題論理における SAT を述語論理へと拡張したものである。限定記号除去とのアナロジーで言えば、SMT は 1 種類の限定記号のみを含む論理式を入力とし、その真偽を判定するものと言える。ただし、SMT は限定記号を取り除くのではなく、様々な代入（$\exists$ なら論理式を満たす代入、$\forall$ なら満たさない代入）を探索することで結果を得る。

SMT の計算量は扱う述語によりけりだが、少なくとも SAT よりは悪いため、理論的にはそれほど高速な解法は望めそうにない。しかし、多くの SMT ソルバーは SAT ソルバーと述語を扱う論理ソルバーの組合せによってできており、近年の SAT ソルバーの高速化を背景に、十分に実用的な速度で動作する実装が開発されてきている。

4　提案手法

それでは、Fregel プログラムの効率化を限定記号除去や SMT ソルバを利用しつつ実現する手法を与える。対象とするのは各 `fregel` 関数である。以下では説明の単純さのためジェネレータは `is` に限る。また、停止条件は「値が変化しなくなると終了」（`Fix`）のみを考える。これ以外にも細かい制限がある。これらについては 4.5 節で議論を行う。

以下、`fregel` 関数が各計算ステップで実行する関数の名前を `step` とする。`step` 関数は図 3 の形式で定義されているとする。各 f_i $(1 \leq i \leq n)$ は隣接頂点から読み出す情報を表す式、p_i $(1 \leq i \leq n)$ はその情報が必要になる条件を表す式、$\bigoplus_i$ $(1 \leq i \leq n)$ はその情報を集約する結合的かつ交換的な演算子、g は隣接頂点の情報をふまえて当該頂点の新しい値を計算する式である。

4.1　通信の削減

まずは通信を削減する方法について考える。Fregel では隣接頂点の情報を読み出す形式でプログラムを記述するが、実際には「次のステップで読み出される値を送信する」プログラムへと翻訳される。そのため、ここでは「送信する必要のない値」すなわち「読み出したとしても計算結果に影響しない値」を特定することを目標とする。以下では n 個ある情報読み出しのうち k 番目のものについて考える。

頂点 u（値 $\dot{u}$）から値を読み出したとしても計算結果に影響しない、という状況を素直に定式化すると以

下となる。

$$\begin{aligned}&\forall v, e, w_1, \ldots, w_n.\\ &\ g(v, w_1, \ldots, w_n) =\\ &\ \ g(v, w_1, \ldots, w_{k-1}, w_k \oplus_k f(e, \dot{u}), w_{k+1}, \ldots, w_n)\end{aligned} \tag{1}$$

ここで、v は値を読み出す頂点の値、e はその頂点と u を繋ぐ辺の値、各 w_i $(1 \leq i \leq n)$ は他の頂点からの情報を集約した値に対応する。つまりこの式は、他の頂点からの情報や値を読み出す頂点の状況によらず、頂点 u の値が不要である状況を表している。

頂点 u から隣接頂点へ値を送る必要があるのは式 (1) が偽となる場合のみである。よって、この真偽を実行時に確認できれば、それをもとに通信を削減することができる。しかしこれは一見現実的でない。一つの方法は、限定記号除去により $\dot{u}$ 以外の変数を除去した式 $\psi(\dot{u})$ を得ることである。これは $\dot{u}$ と基本的な計算のみを含むため実行時にも問題なく真偽を計算できる。具体的には、コンパイラが k 番目の通信の必要性を表す条件 p_k`(prev u)` を p_k`(prev u)` $\wedge \neg\psi$`(prev u)` と変更すればよい。

この素直な定式化は実際にはそれほど有用ではない。例として SSSP を考えると、式 (1) は以下となる。

$$\forall v, e, w.\ \min\{v, w\} = \min\{v, \min\{w, e + \dot{u}\}\}$$

この条件を満たすためには $\dot{u} = \infty$ である必要がある。確かに正しいが、SSSP に関する効率化を完全には捉えていない。

SSSP では頂点の値が更新されていなければメッセージを送る必要はない。これは送信先頂点の値が前回送ったメッセージを十分に考慮しており、二度目は必要ない状況だと理解できる。この状況を先ほどと同様に定式化してみる。いま、頂点 u の前回の値を $\dot{u}$、前々回の値を $\ddot{u}$ とする。

$$\begin{aligned}&\forall v, e, w_1, \ldots, w_n, w'_1, \ldots, w'_n.\\ &\ g(v', w'_1, \ldots, w'_n) =\\ &\ \ g(v', w'_1, \ldots, w'_k \oplus_k f(e, \dot{u}), \ldots, w'_n)\\ &\ \textbf{where } v' = g(v, w_1, \ldots, w_k \oplus_k f(e, \ddot{u}), \ldots, w_n)\end{aligned} \tag{2}$$

この定式化では、式 (1) の場合に比べ、値を読み出す頂点の値 v' は前々回の値 $\ddot{u}$ をもとに計算を行っている、ということが考慮されている。これにより、より正確に通信の必要性を判断できる。具体的な処理は同様であり、限定記号除去を用いて等価な式 $\psi(\dot{u}, \ddot{u})$ を得た後、条件 p_k を更新すれば良い [†6]。

再び SSSP について考える。式 (2) を具体化すると以下となる。

$$\begin{aligned}&\forall v, e, w, w'.\ \min\{v', w'\} = \min\{v', \min\{w', e + \dot{u}\}\}\\ &\ \textbf{where } v' = \min\{v, \min\{w, e + \ddot{u}\}\}\end{aligned}$$

限定記号除去を行うと $\dot{u} \geq \ddot{u}$ が得られる。これは直感とはやや反するが正しい。今回の距離の見積もりが前回より悪ければ、隣接頂点の距離を更新できないため送る必要がない。SSSP の場合、実際には頂点の距離の見積もりが真に悪化することはないので、この条件は「値が変化しない」と等価である。

以上が限定記号除去を用いる方法である。理論上は明瞭だが、式 (2) を用いるとなるとかなりの数の変数が必要となる可能性があり、現実的であるかどうかは疑わしい。以降は代わりに SMT を用いる方法を説明する。

式 (1) や式 (2) は限定記号を $\forall$ しか含んでおらず、一見 SMT ソルバで扱えるように見える。しかし、我々の目的はこの式の真偽を判定することではなく、通信が不要となる条件の効率的な実装 ψ を導出することである。SMT ソルバはこの条件を導出する機能を持たないため、事前にある程度 ψ のテンプレートを用意する必要がある。自然な発想は、前回の値と同じであれば通信を行わない、というものである。これを定式化すると以下となる。

$$\begin{aligned}&\forall \dot{u}, v, e, w_1, \ldots, w_n, w'_1, \ldots, w'_n.\\ &\ g(v', w'_1, \ldots, w'_n) =\\ &\ \ g(v', w'_1, \ldots, w'_k \oplus_k f(e, \dot{u}), \ldots, w'_n)\\ &\ \textbf{where } v' = g(v, w_1, \ldots, w_k \oplus_k f(e, \dot{u}), \ldots, w_n)\end{aligned} \tag{3}$$

式 (2) とは細かい違いがある。まず、$\dot{u}$ と $\ddot{u}$ の違いがなくなっている。これは、前回と値が同じ状況を表現しているためである。さらに、$\dot{u}$ が $\forall$ で限定されている。これは、「どんな $\dot{u}$ であっても、前回と値が同じであれば通信は不要か？」ということを問うているためである。これにより自由変数がなくなり、SMT ソルバで判定が行えるようになる。これが真であれば、

†6 $\ddot{u}$ に対応する値は Fregel の表層言語では得られないが、内部的には「前回と値が変更されていないか」を確認するためにこの値を保持している。

前回と値が同じ頂点はメッセージを送る必要がないことがわかる。なお、SSSP であれば、$\dot{u} = \ddot{u}$ は $\dot{u} \geq \ddot{u}$ の一部であるため、当然真となる。

適当なテンプレートがありさえすれば、SMT ソルバは「前回と値が同じ」以外のケースでも扱うことができる。例えば、各頂点が複数の値を保持している場合、一部の値が等しければ通信をする必要がない可能性は十分ある。このようなケースも、同様に条件を生成し調べれば良い。アプリケーション領域に関する知識があれば、さらに異なる条件（例えば値が大きいなら・小さいなら通信しないなど）を考える可能性もあるだろう。SMT ソルバは限定記号除去に比べて高速であるため、これら可能性やその組合せを探索することも現実的である可能性が高い。とはいえ、アプリケーションを限らない状況では、「前回の値と同じ」という条件は十分に一般的であると考えている。また、この条件は inactive である時に満たされるため、次に述べる頂点の inactive 化と相性が良いという利点もある。

4.2 頂点の inactive 化

次に、頂点を inactive 化する効率化について述べる。頂点が inactive になると、メッセージが届かない限り、その頂点の値は更新されずその頂点からメッセージは送出されない。つまり、頂点の inactive 化はその頂点がメッセージを送る必要がない場合にのみ可能である。よってこの効率化は通信削減の効率化の後に行う。いま、通信削減の効率化を行った後のプログラムが図 3 であるとする。さらに

$$p_*(\texttt{prev}\ u) = \bigwedge_{1 \leq i \leq n} \neg p_i(\texttt{prev}\ u)$$

とする。$p_*(\dot{u})$ は、値 $\dot{u}$ である頂点は何一つメッセージを送る必要がないことを意味する。

頂点を inactive 化しても問題ないのは、「メッセージが届かない限り、頂点の値を更新する必要も、メッセージを送る必要もない」という条件を満たす場合である。これを頂点の値 $\dot{u}$ に対する条件として定式化すると以下となる。ι_i $(1 \leq i \leq n)$ は $\oplus_i$ の単位元であり、メッセージが届かない状況に対応する。

$$p_*(\dot{u}) \wedge (g(\dot{u}, \iota_1, \ldots, \iota_n) = \dot{u})$$

これは限定記号除去や SMT ソルバを用いるまでもなく適切な条件が得られている。コンパイラは、この条件が成り立つならば inactive になる、というコードを挿入すればよい。

アグリゲータがある場合には注意が必要である。アグリゲータの結果はその頂点にメッセージが届かなくても変化しうるので、頂点の inactive 化に際してはその結果を単位元ではなく任意の値としなければならない。例えば k 番目のデータ読み出しが実はアグリゲータであった場合、

$$p_*(\dot{u}) \wedge (\forall w_k.\ g(\dot{u}, \iota_1, \ldots, w_k, \ldots, \iota_n) = \dot{u})$$

という式から w_k を削除し $\dot{u}$ に関する条件を得る必要がある。これに対応し、SMT ソルバの場合にはテンプレートが必要となる。この場合の素直なテンプレートは「通信が不要な場合には inactive になってよい」だろう。これを定式化すると以下となる。

$$\forall \dot{u}, w_k.\ p_*(\dot{u}) \Rightarrow (g(\dot{u}, \iota_1, \ldots, w_k, \ldots, \iota_n) = \dot{u})$$

SSSP の場合、$p_*(\dot{u}) \Rightarrow (\min\{\dot{u}, \infty\} = \dot{u})$ となり、明らかに成り立つ。

4.3 非同期実行

Fregel の `fregel` 関数は、Pregel と同様、BSP に基づいた同期的ば実行が想定されており、それ故 `prev` テーブルの利用が正当化されている。一方で、SSSP であれば、実際には非同期的に実行しても最終的な結果は変わらない。このことを自動的に示すため、まずは以下の補題を準備する。なお証明は難しくないので省略する。

補題 1. 関数 h と h' および 2 項関係 $\preceq$ に対し、以下の条件が成り立つとする。

- h **と** h' **の大小:** 任意の x に対し、$x \preceq h'(x)$ かつ $h'(x) \preceq h(x)$。
- $\preceq$ **の反対称性:** 任意の x と y に対し、$x \preceq y$ かつ $y \preceq x$ ならば $x = y$。
- h **の単調性:** 任意の $x \preceq y$ に対し、$h(x) \preceq h(y)$。

このとき、ある x に対し $h^n(x) = h^{n+1}(x)$（ただし $h^0(x) = x$ かつ $h^{k+1}(x) = h(h^k(x))$）ならば、$h^n(x) = h^n(h'(x))$。 □

いま、補題 1 を以下のように解釈する。まず、h をグラフ全体に対する 1 ステップの完全な処理、h' を

部分的な処理と捉える。つまり、非同期実行は 1 ステップを完全には行わず、処理された頂点と処理されなかった頂点が混在するものとみなしている。このとき、h を繰り返し適用して最終的に得られる結果が h' を適用しても得られるか、を調べたい。なお $\preceq$ は処理の進捗度合いを表しており、最終的に十分大きくなったところで処理が終了することになる。以上の対応をふまえ、`fregel` 関数で記述された処理に補題 1 が適用できれば、非決定的実行が可能であることが確認できる †7。

以上の議論は h や h'、$\preceq$ などがグラフを対象としたものである。しかし、プログラムは各頂点を処理するものとして記述されているため、頂点に関する条件に読み替える必要がある。幸い、グラフに対する比較を「全ての頂点がその関係を満たす」として定義すれば、頂点に対する処理に対して補題 1 が適用できれば、グラフ処理に対しても適用できる。よって、以降は h や h'、$\preceq$ などは頂点を対象としたものとする。このとき、h は「本来あるべきメッセージを全て受け取っての処理」、「h' は本来あるべきメッセージの一部を受け取っての処理」となる。

補題 1 の前提条件を限定記号除去で確認することを考える。この場合、反対称性の確認のためには $\preceq$ の定義が必要となる。反対称性以外の条件を満たす関係を $\preceq$ の定義としても良いのだが、仕様を満たす要素のみを関係づけるとすると比較可能な要素が少なすぎて反対称性を満たさない可能性もあり、有益であるかは疑わしい。ここでは、$\preceq$ は本質的に計算の進行を表すものだという洞察をもとに、$\preceq$ の定義を以下の通り天下り的に与える。

$$x \preceq y \iff \exists w_1, \ldots, w_n.\ g(x, w_1, \ldots, w_n) = y$$

この定義は $x \preceq y$ を「x から何らかのメッセージを受け取って処理を進めると y となる」と読み替えるもので、自然な定義の一つである。以降、式の簡便のため、g はメッセージを 1 つだけ受け取るものとする。多数受け取る場合も全く同様である。

上記 $\preceq$ の定義をもとに補題 1 の条件を確認する。図 3 の一般形を元に書き下すと、以下の 3 条件となる。

h と h' の大小: $x \preceq h'(x)$ は定義より自明なので

$$\forall x, m, m'.\ \exists w.\ g(g(x, m), w) = g(x, m \oplus m')$$

$\preceq$ の反対称性:

$$\forall x, m.\ (\exists w.\ g(g(x, m), w) = x) \Rightarrow (g(x, m) = x)$$

h の単調性:

$$\forall x, m, m'.\ \exists w.\ g(g(x, m'), w) = g(g(x, m), m')$$

「h と h' の大小」では $h'(x)$ と $h(x)$ をそれぞれ $g(x, m)$ と $g(x, m \oplus m')$ で表している。h' はメッセージが十分に届く前の実行を意味することに注意せよ。また、「$\preceq$ の反対称性」と「$\preceq$ の単調性」では、$x \preceq y$ なる y を $g(x, m)$ で表している。これら 3 条件は本質的には限定記号除去で確認が可能である。

例として SSSP を考える。まず関係 $\preceq$ の定義は

$$x \preceq y \iff \exists w.\ \min\{x, w\} = y$$

であり、限定記号を除去すると $x \preceq y \iff x \geq y$ であることが分かる。このことをふまえれば、上記 3 条件は簡単に確認できる。

次に SMT ソルバで条件を確認することを考える。反対称性は以下と等価であるため SMT ソルバでも確認できる。

$$\forall x, m, w.\ (g(g(x, m), w) = x) \Rightarrow (g(x, m) = x)$$

残り 2 つの条件は限定記号のネストがあるため直接は扱えない。この問題を解消するため、十分条件として $w = m'$ の場合を考える。このとき、$\oplus$ の交換性に注意すると、両条件は以下の 1 条件にまとめられる。

$$\forall x, m, m'.\ g(g(x, m), m') = g(x, m \oplus m')$$

この条件は「遅れてやってきたメッセージ m' を反映すると最初から全てのメッセージが揃っていた場合と同じ結果が得られる」ということを主張しており、メッセージの未達にかかわらず処理を行う非同期実行の状況をまさしく捉えている。そのため、この条件によってほとんどの場合は非同期実行可能性が適切に判断できると思われる。

4.4 優先度スケジューリング

2.3 節において、非同期実行が他の様々な最適化を導くことを述べた。この中で、部分グラフ主体処理と高次数頂点分割はこれ以上の解析なしに適用できる。しかし、優先度スケジューリングを行うためには、優

†7 ただし、終了判定は最終的に h（すなわち同期的な 1 ステップ）を実行して結果が変わらないことをもって判断する必要がある。

先度を判断する方法が必要となる。

優先度スケジューリングの基本はより最終結果に貢献するであろう値を優先的に伝播させることである。幸い、非同期実行の定式化に用いてきた関係 $\preceq$ が、値が最終的な結果に近いかどうかの判定に利用できる。この関係で「大きい」ほど最終結果に近いはずで、それ故優先的に伝播すべきである。例えば SSSP であれば「小さい」値ほど優先的に伝播すべき、となる。しかし、この自然なアプローチは実用上常に有用であるわけではない。特に、$\preceq$ が全順序ではなく、推移則を満たさなかったり比較不能な値があったりする場合が問題である。この場合、プライオリティキューを用いるような標準的で効率的な実装が利用できない。

以上の状況をふまえれば、望まれるのは $\preceq$ を拡張した全順序 $\sqsubseteq$、すなわち $a \preceq b \Rightarrow a \sqsubseteq b$ を満たすようなものである。これは、限定記号除去を用いるとしても、SMT ソルバを用いるとしても、全順序 $\sqsubseteq$ のテンプレートなしには与えることができず、またテンプレートがあれば自動的に検証できる。テンプレートとしては、アプリケーションに関する知識が無いとすれば、値の型から分かるよく知られた全順序（数値に対して大小比較など）やその組合せ（組の各要素を辞書式に比較するなど）を網羅的に試すのが適当である。

4.5 拡張と制限

ここまでは、他の頂点のデータの読み出し（すなわち通信）はジェネレータとして `is` を用いたもののみを考えていた。ジェネレータとして何を用いるかは、どのような頂点間で通信するかを規定する。しかし、ここまで議論は、注目している通信以外の通信は誰が何を送っているのか全く分からないという定式化で行っており、複数のジェネレータが混在するとしても問題はない。逆に言えば、例えば同じ値を何度も送るようなプログラムだった場合、同じジェネレータを使っているかどうかを調べることで、通信が不要な条件をより詳しく調べることができるかも知れない。ジェネレータとして全頂点を返すものを用いる場合、すなわちアグリゲータを使う場合についても同様である。論理的には、全ての頂点と接続している頂点を考え、アグリゲータはその頂点との通信と捉えれば良い。ただし、この頂点は独自の値を持たないため、前回送信した値を考慮した通信の削減は意味を持たない。

停止条件は「値が変化しなくなると終了」（`Fix`）のみを考えてきた。これ以外の停止条件を用いたプログラムは `Fix` を用いたものに翻訳できる。大まかには、停止条件の真偽を各頂点が追加の値として保持し、それが真となったら次の計算ステップでは値を変化させなければよい。実際、この変換は Fregel 処理系のコンパイル過程の一つである。

提案手法では `fregel` 関数のみを対象としてきた。他のグラフ処理関数、例えば `gmap` や `gzip` は通信を行わず処理の繰り返しも行わないため提案手法の対象ではない。これらグラフ処理関数による計算を外部のコントロール構造によって繰り返している可能性はあるが、このようなケースの取り扱いについては今後の課題である。

ここまでの説明では、`fregel` 関数は前回の計算ステップの値を保持する `prev` という表のみを用いていたが、実際には今回の計算ステップの値を保持する `curr` という表も利用できる。`curr` を用いたプログラムも `curr` を用いないプログラムへと変換でき、実際 Fregel コンパイラはこれを行っている。よって、理論上は `curr` の利用に問題はない。しかし、この変換の結果、各計算ステップでの関数は内部状態次第で計算内容を完全に切り替えるような複雑なものとなるため、変換後のプログラムに対して効率化が可能だと判定できるかどうかは定かでない。変換を経ずに直接最適化を行う方法は今後の課題である。

5 実装と実験

提案手法の可能性を探るため、前述の効率化を行うよう既存の Fregel コンパイラを改造し、効率化の効果を確認した。

5.1 実装の方針

実装の簡便のため以下の方針で臨んだ。

まず、既存実装の利用が難しいと思われる限定記号除去は諦め、SMT ソルバを用いる手法のみを実装し

表 1　実験用グラフデータ

名前	頂点数	辺数
WebBerkStan	685, 230	7,600,595
RoadNet-PA	1,088,092	1,541,898
Rand1M/10M	1,048,576	10,485,760

表 2　SSSP プログラムに対する最適化の効果（単位：秒）

データ名	最適化前	通信削減	inactive 化	手書き
WebBerkStan	401.3	54.2	47.3	24.2
RoadNet-PA	228.8	69.6	53.5	29.0
Rand1M/10M	29.5	10.7	9.9	4.3

た。SMT ソルバによる手法は限定記号除去による手法に比べ効率化の能力が低いため、SMT ソルバでも十分な効率化が可能であれば、提案手法の有用性は十分であると言える、との判断である。

次に、現時点での Fregel コンパイラのバックエンドである Giraph と Pregel+が非同期実行をサポートしていないため、非同期実行に関する効率化の実装は諦めた。非同期実行の効果に関しては既に詳しく調べられており [10, 11, 28]、改めて調査する価値は低い。また、効率化手法としては通信量削減等の延長線上にあり、本質的な困難があるとは考えがたい。

さらに、Fregel コンパイラの中間表現ではなく抽象構文木に対する変換として効率化を実装した。これは、抽象構文木に対してあれば前節の議論が全くそのまま使え簡便であり、また中間表現に変換する過程でもたらされるコードの複雑性が効率化可能性の判定に悪影響を与える可能性を危惧したためである。この結果、`Fix` 以外の停止条件や `curr` などは扱えなくなってしまっている。

SMT ソルバとしては、Z3 ソルバ（version 4.3.2）を用いた。実装に際して必要な機能はほとんどが既に提供されていた。ただし、min や max の単位元である ∞ と $-\infty$ が必要となるため、数値（特に整数）に関してはこれらを含む直和型として拡張し、各演算を適切に再定義する Z3 プログラムを記述した。

5.2　実験の概要

図 1 に示した SSSP の Fregel プログラムに対して最適化を適用し、その効果を確認した。最適化の効果を詳しく見るため、(1) 最適化前 (2) 通信の削減のみを適用したもの (3) 通信の削減と inactive 化を適用したもの (4) 手書きのプログラム、の 4 種類を用意し、それぞれの 5 回の実行の平均時間算出した。なお、実行時間はグラフデータの読み出し・書き込みをはじめとする初期化・終了処理の時間を含まない。

実験環境は 16 台の PC がギガビットイーサで繋がった PC クラスタである。各計算機は CPU が Intel Core i5（9 台は Core i5-2500、残り 7 台は i5-760）、メモリ 8 GB 、そして 128 GB の SSD を積んでいる。OS は Ubuntu 14.04.3 LTS である。Fregel のバックエンドとしては Giraph を用いており、Giraph 1.2、Hadoop 1.2.1、Java 1.8.0_121 からなる。全ての実験で 16 ワーカーで実行をしている。

実験用のグラフデータは表 1 に示す 3 つを用いた。WebBerkStan と RoadNet-PA はスタンフォード大学が提供しているグラフデータ [†8] であり、前者は web グラフ、後者は道路ネットワークである。Rand1M/10M は頂点間をランダムに繋いだグラフである。

5.3　実験結果

以下、実験結果について述べる。まず、SMT ソルバによる効率化可能性の判定は一瞬で終了し、通信の削減、及び inactive 化が可能であることが判断された。次に、各プログラムの実行時間を表 2 に示す。データにもよるが、最適化前は手書きのコードに比べて数倍から十数倍程度実行性能が悪かった。これに対し、通信削減、inactive 化両方を行うことで、2 倍程度の遅さにまで改善しており、最適化の効果は非常に大きい。特に通信削減の効果は大きく、Fregel の自然な記述がもたらす無駄な通信が大きなコストになっていたことが示唆される。また、inactive 化の効果も、通信削減ほどではないものの、10 ～ 20% 程度の改善であり小さくはない。残る差は Fregel コンパイラの生成するコードに含まれる細かいオーバーヘッドだと思われるが、精査はしていない。

†8　https://snap.stanford.edu/data/

6 関連研究

頂点主体並列処理に関する関連研究については 2.3 節で既に述べた。本研究は、これら研究で提案されてきた頂点主体並列処理の効率化方法を背景に、効率化を適用したプログラムを自動的に、しかも可能な限りアプリケーションに関する知識を用いずに得る手法を提案した。

限定記号除去に代表される高度な制約解消器を並列プログラムの生成に用いること自体は以前から行われている。典型的な利用方法は、例えばステンシルコードのように、ネストしたループ構造をもつ配列操作プログラムに対し、適切なループ変換の方法を模索したり、その際の無駄な計算・通信などを取り除いたりするものである [1,8,17]。特にプログラムを線形不等式系によって特徴付ける多面体モデルを用いるアプローチでは、線形不等式系に対する限定記号除去操作が知られているため、これら制約解消器が解析の基礎として用いられている。また、これ以外にも、分割統治並列プログラムのためのプログラム断片を得る手段として用いた例もある [15,16,21,23]。SMT ソルバー等がこのようなプログラム生成の手段として有用であることは、近年では広く認識されつつある [9,24]。

本研究ではこれら手法を大規模グラフ処理という極めてイレギュラーな並列処理の効率化に用いた。イレギュラーは並列処理では、その計算の構造を多面体モデルなどで捉えることが難しいため、一般にプログラムの解析は困難となる。例えば、頂点主体並列処理を計算ステップを繰り返す `while` ループと全ての頂点を操作する `for` ループからなる 2 重ループだと思えば、非同期実行可能性の検査はループ入れ子の実行順序の変換であり、まさに制約解消器などがよく用いられる題材である。しかし、グラフの構造に依存したデータ依存性や計算結果に依存した停止条件などのため、このループ変換は難しい。これに対し、本研究では、頂点主体並列処理プログラムの構造に着目することで補題 1 を導入し、これにより現実的な検査方法を提案している。以上の議論からも見て取れるように、本研究の主要な新しさは、既存技術をもとに頂点主体並列処理に対する解析・変換手法を与えた点である。

本研究では Fregel でプログラムを記述するという方針をとった。Fregel の特徴の一つは宣言的であることである。このことは宣言的な記述を要求するソルバーの利用を容易にしている。しかし、宣言的であることは本研究の顕著な特徴ではない。まず、非宣言的なプログラムであったとしても、多少の現実的な仮定の下で、ソルバーの利用は多少困難にはなるものの可能ではある。また、Fregel 以外にも宣言的に並列分散グラフ処理を記述する枠組みはいくつもあり [3,7,18,19,22]、特に頂点主体並列処理に限っても GraphX [7] という完成度の高いシステムがある。GraphX は Scala 上の並列グラフ処理フレームワークで、MapReduce のようにデータの流れを高階関数を用いて記述することを特徴としており、その一環として Pregel API も提供している。

本研究に特に関連する Fregel の特徴は、処理が BSP モデルに基づいている点、そしてメッセージや頂点処理の必要性、頂点の走査順序などを全く指定しないという点である。まず前者の点により、Fregel プログラムの挙動は決定的であり、並列実行の計算結果は逐次実行の結果と必ず一致する。これは他の宣言的な枠組みの多くが、出力の満たすべき条件を論理的に特定することを主眼に置いており、その条件を満たす出力が複数ある場合について非決定性を許しているのとは異なる。非決定性を許す方針の方がより宣言的であり、また本発表でも論じたように効率化も行いやすいが、現実的にはこれはプログラムの構築、特にデバッグを難しくしてしまう。また、宣言的な枠組みであっても、効率をプログラマが改善するための何らかの記述を前提としていることが多い。例えば、GraphX の Pregel API は本家 Pregel と同様、送信すべきメッセージや inactive にすべき頂点などを明示的に指定するものとなっている。Coordinated Linear Meld [22] はグラフ処理のための論理型言語だが、データの分割やデータ処理の優先順位を追加で指定することで効率を高めるようになっている。

本研究と最も関連の深い研究は、Elixir [18,19] と Distributed SociaLite [22] である。Elixir は出力グラフが満たすべき性質の論理的な記述からグラフを分散並列処理するプログラムを自動導出するシステムで

ある。特に、次の計算ステップでグラフ全体を走査することを避けるため、走査する必要のある頂点集合を特定するプログラムを SMT ソルバの助けを借りつつ合成している点が、本研究における inactive 化とよく似ている。なお、頂点がメッセージを送る条件や頂点を走査する優先順位などはユーザが明示的に指定している。Distributed SociaLite は datalog をもとにした宣言的なグラフ処理言語である。SSSP のような計算を加速するため、計算がある種の単調性を満たすことをユーザが保証することで、Δ-stepping [14] の一般化を適用している。以上のように、本研究は Elixir や Distributed SociaLite に比べ、より少ない情報から、より多くの最適化を自動的に適用することを目指しており、頂点走査の優先順位や単調性など、Elixir や Distributed SociaLite ではユーザが指定していた部分も自動導出している。同時に、Δ-stepping のような既存アルゴリズムを適用するのではなく、そのようなアルゴリズムが自動的に導出されるような効率化手法を目指し、非同期実行や優先度スケジューリングの導入方法を提案している。また、Elixir も Distributed SociaLite もその仕様記述の性質上非決定的実行を想定しているが、Fregel があえてより最適化の難しい BSP モデルから出発している点も小さくない差異である。しかし、Elixir も Distributed SociaLite も頂点主体に限らないグラフ処理をサポートしており、より表現力は高い。本研究のアプローチがより広いグラフ処理へと適用できるかの議論は今後の課題である。

7 まとめと今後の課題

本発表では、頂点主体並列処理を効率化する手法を提案した。限定記号除去・SMT ソルバーを用いることで、通信量の削減や頂点の inactive 化はもちろん、非同期的実行や優先度スケジューリングも自動的に導入できることを示した。SMT ソルバーを用いた予備的実装は現実的な時間で最適化を行えており、最適化の効果も大きいことが観察できている。

現状の実装は予備的なものであり、大きく複雑なプログラムの取り扱いを前提にしたものではない。本格的な実装を行い、複雑なプログラムにも提案手法が有効であるか否かを調べるのは今後の課題である。また、同様の手法は本論文で述べた以外の高度な効率化にも有用である可能性がある。そのような可能性も今後模索してゆくことを考えている。

謝辞 提案手法と関連する研究についての情報を提供してくださった高知工科大学の佐藤重幸氏に感謝する。本研究の一部は、JSPS 科研費 JP26280020、JP15K15974 の助成を受けている。

参 考 文 献

[1] Bondhugula, U., Baskaran, M. M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model, *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, Hendren, L. J.(ed.), Lecture Notes in Computer Science, Vol. 4959, Springer, 2008, pp. 132–146.

[2] Caviness, B. F. and Johnson, J. R.(eds.): *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Springer Vienna, Vienna, 1998.

[3] Cruz, F., Rocha, R., and Goldstein, S. C.: Declarative coordination of graph-based parallel programs, *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Asenjo, R. and Harris, T.(eds.), ACM, 2016, pp. 4:1–4:12.

[4] de Moura, L. M. and Bjørner, N.: Satisfiability modulo theories: introduction and applications, *Commun. ACM*, Vol. 54, No. 9(2011), pp. 69–77.

[5] 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 2016, Nara, Japan, September 18-22, 2016*, Garrigue, J., Keller, G., and Sumii, E.(eds.), ACM, 2016, pp. 200–213.

[6] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Thekkath, C. and Vahdat, A.(eds.), USENIX Association, 2012, pp. 17–30.

[7] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I.: GraphX: Graph Processing in a Distributed

Dataflow Framework, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, Flinn, J. and Levy, H.(eds.), USENIX Association, 2014, pp. 599–613.

[8] Größlinger, A., Griebl, M., and Lengauer, C.: Quantifier elimination in automatic loop parallelization, *J. Symb. Comput.*, Vol. 41, No. 11(2006), pp. 1206–1221.

[9] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R.: Synthesis of loop-free programs, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Hall, M. W. and Padua, D. A.(eds.), ACM, 2011, pp. 62–73.

[10] Han, M. and Daudjee, K.: Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems, *PVLDB*, Vol. 8, No. 9(2015), pp. 950–961.

[11] Liu, Y., Zhou, C., Gao, J., and Fan, Z.: GiraphAsync: Supporting Online and Offline Graph Processing via Adaptive Asynchronous Message Processing, *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*, Mukhopadhyay, S., Zhai, C., Bertino, E., Crestani, F., Mostafa, J., Tang, J., Si, L., Zhou, X., Chang, Y., Li, Y., and Sondhi, P.(eds.), ACM, 2016, pp. 479–488.

[12] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M.: Distributed GraphLab: A Framework for Machine Learning in the Cloud, *PVLDB*, Vol. 5, No. 8(2012), pp. 716–727.

[13] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: a system for large-scale graph processing, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Elmagarmid, A. K. and Agrawal, D.(eds.), ACM, 2010, pp. 135–146.

[14] Meyer, U. and Sanders, P.: [Delta]-stepping: a parallelizable shortest path algorithm, *J. Algorithms*, Vol. 49, No. 1(2003), pp. 114–152.

[15] Morihata, A. and Matsuzaki, K.: Automatic Parallelization of Recursive Functions Using Quantifier Elimination, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, Blume, M., Kobayashi, N., and Vidal, G.(eds.), Lecture Notes in Computer Science, Vol. 6009, Springer, 2010, pp. 321–336.

[16] Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Ferrante, J. and McKinley, K. S.(eds.), ACM, 2007, pp. 146–155.

[17] Pouchet, L., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., and Vasilache, N.: Loop transformations: convexity, pruning and optimization, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Ball, T. and Sagiv, M.(eds.), ACM, 2011, pp. 549–562.

[18] Prountzos, D., Manevich, R., and Pingali, K.: Elixir: a system for synthesizing concurrent graph programs, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Leavens, G. T. and Dwyer, M. B.(eds.), ACM, 2012, pp. 375–394.

[19] Prountzos, D., Manevich, R., and Pingali, K.: Synthesizing parallel graph programs via automated planning, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, Grove, D. and Blackburn, S.(eds.), ACM, 2015, pp. 533–544.

[20] Quamar, A., Deshpande, A., and Lin, J. J.: NScale: neighborhood-centric large-scale graph analytics in the cloud, *VLDB J.*, Vol. 25, No. 2(2016), pp. 125–150.

[21] Sato, S. and Iwasaki, H.: Automatic parallelization via matrix multiplication, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Hall, M. W. and Padua, D. A.(eds.), ACM, 2011, pp. 470–479.

[22] Seo, J., Park, J., Shin, J., and Lam, M. S.: Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis, *PVLDB*, Vol. 6, No. 14(2013), pp. 1906–1917.

[23] Smith, C. and Albarghouthi, A.: MapReduce program synthesis, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Krintz, C. and Berger, E.(eds.), ACM, 2016, pp. 326–340.

[24] Srivastava, S., Gulwani, S., and Foster, J. S.: Template-based program verification and program synthesis, *STTT*, Vol. 15, No. 5-6(2013), pp. 497–518.

[25] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J.: From "Think Like a Vertex" to "Think Like a Graph", *PVLDB*, Vol. 7, No. 3(2013), pp. 193–204.

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

[27] Verma, S., Leslie, L. M., Shin, Y., and Gupta, I.: An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing, *PVLDB*, Vol. 10, No. 5(2017), pp. 493–504.

[28] Xie, C., Chen, R., Guan, H., Zang, B., and Chen, H.: SYNC or ASYNC: time to fuse for distributed graph-parallel computation, *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, Cohen, A. and Grove, D.(eds.), ACM, 2015, pp. 194–204.

[29] Yan, D., Cheng, J., Lu, Y., and Ng, W.: Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, Gangemi, A., Leonardi, S., and Panconesi, A.(eds.), ACM, 2015, pp. 1307–1317.