情報処理学会論文誌　コンピューティングシステム　Vol.9　No.1　30–40 (Mar. 2016)

Pregelグラフ処理系におけるメッセージ配送最適化

上野 晃司[1,3,a)]　鈴村 豊太郎[2,3,b)]　松岡 聡[1,3,c)]

受付日 2015年7月31日, 採録日 2015年11月22日

概要：ソーシャルネットワークやWebグラフなど大規模グラフ解析の需要は近年高まっている．Pregelは大規模グラフの並列分散処理系の中でも最も簡単にグラフアルゴリズムを記述できるプログラミングモデルの1つであるが，処理速度の遅さが問題であった．本論文ではSendAllというメッセージ配送最適化手法を提案する．SendAllはすべての隣接頂点に同じメッセージを送る場合にしか適用できないという制約はあるが，計算処理を少し変えるだけで多くのグラフアルゴリズムに適用可能な最適化手法である．TSUBAME2.5を使った性能評価ではSendAllを適用しない場合に比べて，PageRankで1.80倍～4.04倍，幅優先探索で1.76倍～4.76倍に高速化した．また，通信データ量はR-MATグラフに対して32ワーカで動作させた場合Combinerを使った場合と比較してPageRankで45%削減した．

キーワード：グラフ処理，分散メモリ，Pregel

Optimized Message Communication Method for Pregel Graph Processing System

Koji Ueno[1,3,a)]　Toyotaro Suzumura[2,3,b)]　Satoshi Matsuoka[1,3,c)]

Received: July 31, 2015, Accepted: November 22, 2015

Abstract: In this several years, demand of large scale graph analysis has been bigger than before. Pregel is one of the most simple and straightforward programming model for distributed parallel graph processing. However, the problem of Pregel is low performance. We propose a new efficient message handling technique, called SendAll. Although SendAll can be applied only when a vertex sends the same message to all neighbor vertices, SendAll can be applied to many graph algorithms with a little change in the program. The result of our performance evaluation on TSUBAME2.5 shows that PageRank gets 1.80–4.04 times faster and Breadth-first search gets 1.76–4.76 times faster with SendAll. SendAll can also reduce the communication data volume. When we compute PageRank for R-MAT graph with 32 workers, the amount of data transferred between workers is reduced by 45 percent.

Keywords: graph processing, distributed memory, pregel

1. はじめに

グラフとは頂点とそれらを結ぶエッジの集合で表されたデータ構造である．人を頂点，人と人との関係をエッジで表したソーシャルネットワーク，Webサイトを頂点，ハイパーリンクをエッジで表したWebグラフ，脳神経細胞とそのつながりを表したネットワークなど，実世界には大規模なグラフが多くあり，その解析の需要は日に日に高まっている．

Pregel [1] はGoogleにより提唱された分散並列グラフ処理系である．コモディティマシンのクラスタ上で大規模なグラフを分散並列処理することを目的として作られた．Pregelではグラフデータを分割し，複数の計算機ノードに分散配置されるので，ノード数の許す限り，巨大なグラフデータを処理することができる．

[1] 東京工業大学
Tokyo Institute of Technology, Meguro, Tokyo 152–8550, Japan
[2] トーマス・J・ワトソン研究所
IBM T.J. Watson Research Center, NY 10598, USA
[3] JST CREST, Chiyoda, Tokyo 102–0076, Japan
[a)] ueno.k.ac@m.titech.ac.jp
[b)] suzumura@acm.org
[c)] matsu@acm.org

大規模データの分散並列処理では，MapReduce [2] 処理系が一般に最も普及している．しかし，グラフ処理は，データアクセスのランダム性が強く，また，反復回数が多い処理なので，反復ごとに全データにアクセスしなければならない MapReduce はグラフ処理が得意ではない．Pregel は必要なデータにだけアクセスして少ないコストで反復処理できるので，グラフ処理を比較的高速に行うことができる．

数あるグラフ処理系の中で Pregel は，ユーザ定義関数 1 つだけで様々なグラフアルゴリズムを記述でき，頂点を中心にとらえてプログラムを記述する vertex-centric という点が直観的で分かりやすい．プログラムの記述性では Pregel は大きなメリットがある．しかし，その反面，処理が遅いという問題をたびたび指摘されている [3], [4]．

たとえば，Tian ら [3] は，Pregel のような vertex-centric モデルではなく，より高速に計算が可能な graph-centric モデルを提案し，連結成分分解で graph-centric と vertex-centric では 10〜63 倍の実行時間差があったと報告している．また，Shun ら [4] は，並列グラフ処理フレームワーク Ligra を提案しているが，Pregel 実装の 1 つである GPS [5] との比較で，40CPU コアを搭載したマシンでの Ligra による実行は，4 コアのノード 30 台のクラスタ（合計 120 コア）での GPS による実行より，4 倍以上高速だったと報告している．

ただし，前者の graph-centric は，Pregel のような簡単なプログラミングモデルではない．また，後者の Ligra は複数ノードによる分散処理に対応していないので計算できる問題のサイズが限られる．Pregel グラフ処理系は，プログラミングが簡単，かつ，ノード台数があれば計算可能な問題サイズにほぼ制限がないので，Pregel グラフ処理系のパフォーマンス改善は大きな意義を持つ．

本論文では，Pregel のノード間通信である頂点間メッセージの配送に着目し，効率良くメッセージを配送する手法 SendAll を提案する．SendAll は Pregel のプログラミングモデルを拡張はするものの，ほぼ維持したまま性能を引き上げることが可能である．以下，本論文の貢献を示す．

(1) Pregel における効率の良いメッセージ配送手法 SendAll の提案
(2) SendAll を使ったアプリケーション実装の提示
(3) 性能評価の結果，従来手法に比べて最大 5 倍程度まで実行速度を向上

以降，2 章では Pregel とその実装の 1 つである XPregel，また，Pregel におけるメッセージ配送の実装方法について，3 章では，提案手法である SendAll について，4 章で性能評価とその結果，5 章で関連研究，6 章でまとめについて述べる．

2. Pregel グラフ処理系

この章では，まず，Pregel [1] プログラミングモデルと，その実装の 1 つである XPregel [6], [7] を説明し，次に Pregel 処理系におけるメッセージ配送について説明する．

2.1 Pregel

Pregel [1] のプログラミングモデルはバルク同期並列 [8] から着想を得ている．Pregel ではスーパステップと呼ばれるステップの反復で計算する．スーパステップでは，各頂点に対してユーザ定義関数 compute() を実行する．複数の頂点に対するユーザ定義関数の処理は並列して実行される．ユーザ定義関数では，メッセージを他の頂点に送信（SendMessage）したり，また，1 つ前のスーパステップで送信されたメッセージを受け取ったり，頂点に関連付けられたデータを更新したりなどができる．次のスーパステップに移行するときに全体で同期をとるので，複数のスーパステップが同時に実行されることはない．

頂点は，アクティブ状態と非アクティブ状態があり，最初のスーパステップ開始時は，すべての頂点がアクティブ状態に設定される．アクティブ状態の頂点は，各スーパステップで実行されるユーザ定義関数から非アクティブ状態にすることができる．非アクティブ状態の頂点にメッセージが送信されるとその頂点はアクティブ状態になる．すべての頂点が非アクティブ状態，かつ，メッセージが 1 つも送信されない場合に限り，スーパステップの反復が終了する．

Pregel ではグラフを分割し，複数のワーカで保持する．頂点は必ずいずれか 1 つのワーカに所有される．頂点は，通常その頂点から出ているエッジ（出力エッジ）を持っている．以下，頂点 u から頂点 v へのエッジがあった場合，このエッジ (u,v) を頂点 u の出力エッジ，頂点 v の入力エッジと呼ぶ．

2.2 XPregel

我々は，Pregel グラフ処理系のプログラミングモデルを拡張し XPregel を開発した．XPregel は，IBM Research の開発した並列分散処理の記述が容易な言語 X10 [9] を使用している．XPregel はオープンソースで開発している大規模グラフ処理ライブラリ ScaleGraph [6], [7] の中核をなすグラフ処理系である．

Pregel では，計算開始時にファイルからグラフデータなどの入力データを読み込み，計算終了時に計算結果をファイルに出力するようになっている．Pregel がファイルから入力データを読み込み，計算結果をファイルに出力するまでを 1 回の計算とすると，XPregel ではこの部分が拡張され，複数の計算をオンメモリで処理することができるようになっている．これにより，パフォーマンスが向上しているのはもちろん，複雑なグラフアルゴリズムがより記述しやすくなっている．

また，XPregel は計算機ノード内の並列化をスレッド，

```
val xpgraph = XPregelGraph.make(...); // グラフ読み込み
xpgraph.updateInEdge();
xpgraph.iterate[Double,Double](
(ctx :VertexContext[Double,Double,Double,Double]) => {
  val value :Double;
  if(ctx.superstep() == 0) {
    value = 1.0 / ctx.numberOfVertices();
  }
  else {
    var sum :Double = 0.0;
    for(mgs in ctx) // 受信メッセージループ
      sum += mgs;
    value = 0.15 / ctx.numberOfVertices() + 0.85 * sum;
  }
  ctx.setValue(value);
  ctx.sendMessageToAllNeighbors(value / ctx.outEdgesId
      ().size());
},
null,
(superstep :Int, aggVal :Double) => superstep < 30
);
```

図 1　XPregel による PageRank の実装
Fig. 1　PageRank implemented in XPregel.

ノード間の並列化を MPI により行っている．以下, XPregel において，MPI 並列時の 1 MPI プロセスをワーカと呼ぶ．

図 1 に XPregel による PageRank 実装を示す．変数 ctx は VertexContext であり，これは XPregel のフレームワークサービスを提供する API で，メッセージの送受信や，頂点の状態更新などの機能にアクセスするためのインタフェースである．sendMessageToAllNeighbors() はすべての隣接頂点に同じメッセージを送る関数である．PageRank は Web ページなどにおいてリンク構造を使ってページの重要度を計算するアルゴリズムである．各頂点はページであり，各ページは固有の得点（value）を持っている．各スーパステップでページの得点をそれぞれの隣接頂点に分配するが，その分配する得点をメッセージとして送信している．

2.3　Pregel メッセージ配送

Pregel において頂点間のデータのやりとりは，メッセージの送受信で行う．このメッセージの配送は Pregel 処理系が行う最も重要な処理である．Pregel の並列分散処理において，グラフは複数のワーカに分割される．メッセージが別のワーカに属する頂点に送信されるとき，ワーカ間のデータ通信が発生する．このワーカ間のデータ通信やメッセージデータのやりとりの高速化は Pregel におけるグラフ処理全体の処理高速化に非常に重要となる．

2.3.1　通常のメッセージ配送手法

メッセージ配送の実装にはいくつか方法があるが，ここでは XPregel の実装方法を説明する．メッセージ配送では，全対全（all-to-all）の通信が必要になる．全対全通信は，通信負荷が最も高い通信の 1 つで，スーパコンピュータのようなノード間ネットワークの比較的高速なシステムにおいても，ボトルネックとなりやすい通信である．XPregel では，全対全通信を効率良く行うため，MPI の全対全通信

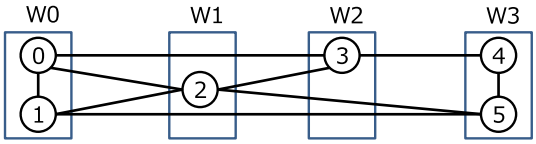


図 2　4 ワーカで 6 頂点のグラフを分割する例
Fig. 2　Graph partitioning example for 6 vertices graph with 4 workers.

関数を呼び出している．

Pregel プログラミングモデルにおいて，ユーザがメッセージの送信に使う API が，SendMessage() 関数である．SendMessage() はユーザ定義関数 compute() から呼ぶことができ，行き先頂点 ID とメッセージの値を引数として渡す．SendMessage() で与えられた送信メッセージを行先頂点に配送するときの，各ワーカでの処理の流れを以下に示す．

N1　SendMessage() が呼び出され，メッセージは行き先ワーカごとのバッファに入れられる．
N2　全頂点への compute() 呼び出しが終了後，メッセージを MPI で送信するため，1 本の配列にマージする．
N3　MPI_Alltoallv で通信．
N4　受け取ったメッセージ配列のメッセージを行き先頂点ごとにまとめる必要があるため，行き先頂点 ID でソート．

N4 のソートは計算量の大きい処理であり，XPregel ではバケツソートを数段階で行うマルチレベルソートで実装されている．

2.3.2　Combiner によるメッセージデータ量削減

Pregel ではワーカ間の通信データ量を削減するため，Combiner という仕組みを備えている．combiner() は行先が同じ頂点である複数のメッセージをマージする方法を定義したユーザ定義関数である．ユーザが combiner() を提供することで，処理系はワーカ間でデータを通信する前に，combiner() を使ってメッセージ数を減らすことができる．

Combiner の動作原理を**図 2** のように 6 頂点の簡単なグラフを 4 ワーカで分割した例を用いて説明しよう．図 1 にあるアルゴリズムで PageRank を計算する場合, combiner() にはメッセージの値を足し合わせて 1 つのメッセージにする関数を使用することができる．それにより，たとえば，頂点 0 から頂点 2 に送られるメッセージと，頂点 1 から頂点 2 に送られるメッセージはマージすることができ，通常のメッセージ配送ではワーカ 1 からワーカ 2 に 2 つメッセージが送られるところが，Combiner を使うと 1 つのメッセージになる．

XPregel では，通常のメッセージ配送の処理の流れの N2 と N3 の間に，以下の処理を挟むことで Combiner によるメッセージ数削減を実現している．

C1 行き先頂点 ID でソート

C2 Combiner を実行しメッセージ数を削減

ただし，Combiner を使用すると，計算量は増えてしまう．特に，C1 のソートは計算量の大きい処理であり，これによる計算量の増加は問題である．Pregel はスパコンと比較してノード間通信速度の遅いコモディティクラスタ向けに提案されたフレームワークである．スパコンに比べて通信速度の遅いコモディティクラスタの場合，C1 のソートにより計算量が増加しても通信データ量の削減で全体の速度を向上させることができる．しかし，通信速度の速いスパコンの場合，通信データ量が削減されてもそこで得られる時間短縮は少なく，計算量の増加による時間増加の方が大きくなってしまう．つまり，ノード間通信速度の速いスパコンでは Combiner はあまり効果的ではない．また，combiner() を定義できるかどうかは，アプリケーションに依存するので，すべてのアプリケーションで combiner() が使用できるわけではない．

2.4　解決すべき問題

Combiner はメッセージデータ量を削減し，処理を高速化することが狙いだったが，計算量が増加するので，実際には高速化に使用するのは難しい．しかし，通常のメッセージ配送では，通信データ量が大きく，効率が良いとはいえない．メッセージデータ量の削減と，高速な計算が両立するメッセージ配送手法が必要とされている．そこで，本論文では，この 2 つを両立する SendAll という手法を提案する．

3.　SendAll による最適化

SendAll は，すべての隣接頂点に同じメッセージを送る場合に適用可能な最適化手法である．この章では，まず，SendAll の概要とアルゴリズムの説明を行い，次に，計算量やデータ量の性質，SendAll の適用可能範囲を広げるための API の追加について説明する．

3.1　基本動作

隣接頂点はエッジで接続されている頂点のことである．グラフ処理では，すべての隣接頂点に同じメッセージを送ることが多い．たとえば，幅優先探索，連結成分分解，エッジの重みなしの PageRank などのアルゴリズムの典型的な実装では，すべての隣接頂点に同じメッセージを送る．SendAll は全隣接頂点に同じメッセージを送る場合に適用可能な最適化であるが，一部の頂点だけがメッセージを送ることも可能である．ただし，メッセージを送る頂点はすべての隣接頂点にメッセージを送る必要がある．

API としては図 1 の PageRank の実装例でも使用している sendMessageToAllNeighbors 関数を使用する．この関数が使われた場合，すべての隣接頂点に同じメッセージを送るので，本提案手法が適用可能となる．

まず，SendAll がどのようにしてメッセージデータ量を削減するのかについて，図 2 のグラフで PageRank を計算する場合を例に説明する．頂点 2 からワーカ 0 にある 2 つの隣接頂点に送られるメッセージに着目する．通常のメッセージ配送では，この 2 つの隣接頂点（頂点 0，1）宛てのメッセージは 2 つのメッセージとして送信される．このとき，別頂点宛てではあるが，同じメッセージが 2 つ送られることになるので，無駄がある．そこで，メッセージを 2 つ送ることはせず，まず，メッセージを 1 つ送信し，ワーカ 0 で頂点 0，1 に配送することで，メッセージデータ量の削減を可能とする．

次に，高速な計算の実現方法であるが，SendAll では，ステップ N4 にあるようなメッセージを行き先頂点 ID でソートするようなことはしない．行き先ワーカで各頂点に配送するときの処理は，入力エッジ情報を用いて行うが，全入力エッジを見て，その頂点宛てのメッセージが届いているか確認し，届いていたら読み取るというような動作をする．詳しいアルゴリズムは後述するが，このアルゴリズムの利点は，ワーカ間通信で受信したメッセージ配列にはいっさい手を加えずに，リードアクセスだけで，行き先頂点ごとのメッセージを取得可能な点である．ただし，このリードアクセスはメモリへのランダムアクセスとなるので，CPU のキャッシュが効きにくく，コストの大きなリードアクセスとなる．しかし，メッセージのソートよりはコストの小さい処理なので，高速化が期待できる．

3.2　アルゴリズム

SendAll によるメッセージ送信の起点となるのが，sendMessageToAllNeighbors 関数である．SendAll によるメッセージ送信の各ワーカにおける処理アルゴリズムを以下に示す．

S1 sendMessageToAllNeighbors() の呼び出しで，メッセージはあらかじめ頂点数分のメッセージを格納できるように用意されたバッファ Mtmp に入れられる．このとき，ビットマップ Btmp のフラグを立てる．ビットマップとは各フラグを 1 ビットで表したデータである．Btmp は送信メッセージがあるかどうかを表すビットマップである（ある場合は 1，ない場合は 0）．

S2 全頂点への compute() 呼び出しが終了後，各頂点からのメッセージを，そのワーカに送信すべきかどうかを保持するビットマップ BhasEdge を使って，行先ワーカごとに送信するメッセージ Msend，Bsend を抽出する．BhasEdge は頂点ごとワーカごとにそのワーカへの入力エッジがあるかどうかをフラグとして，ビットマップにしたものであり，あらかじめ作成しておいたものである．Bsend はメッセージがどの頂点から送信されたかを表すビットマップとなる．

S3 Msend，Bsend を Alltoall で通信して，Mrecv，Brecv

```
引数
myId : 送信元頂点のID
mes : 送信メッセージ

def sendMessageToAllNeighbors(myId, mes) {
  Mtmp(myId) = mes;
  Btmp(myId) = 1;
}
```

図 3　sendMessageToAllNeighbors 関数
Fig. 3　sendMessageToAllNeighbors function.

```
引数
Mtmp, Btmp : 入力となるメッセージとフラグ
BhasEdge : 各ワーカへの入力エッジフラグ
Mrecv, Brecv : Alltoall で受け取ったデータ（出力）

def sendAllExchange(Mtmp, Btmp, BhasEdge, Mrecv, Brecv)
      {
  var offset = 0;
  for(w in 0..(numWorkers-1)) {
    send_counts(w) = 0;
    for(i in 0..(Btmp.numWords()-1)) {
      val word = Btmp.word(i) & BhasEdge.word(i)
      Bsend.word(Btmp.numWords()*w + i) = word;
      send_counts(w) += popcount(word);
    }
    for(i in (0..Mtmp.size()-1)) {
      if(Bsend(i) != 0) Msend(offset++) = Mtmp(i);
  }}
  Alltoall(Bsend, Brecv)
  Alltoall(send_counts, recv_counts)
  Alltoallv(Msend, send_counts, Mrecv, recv_counts)
}
```

図 4　SendAll のワーカ間通信アルゴリズム
Fig. 4　Worker communication algorithm in SendAll.

を受け取る．

S4 受け取り側の入力エッジデータを使って，各頂点に届いたメッセージを集める．

S1 の処理を図 **3**，S2〜S3 の処理を図 **4**，S4 の処理を図 **5** にそれぞれ疑似コードを示す．アルゴリズム中，“変数名 (i)” は配列 i 番目の要素を表す．ただし，Brecv(i) はビットマップ Brecv の i 番目のビット，Brecv.word(i) は Brecv の i 番目のワードを返す．ワードは複数ビットのかたまりであり，XPregel では 1 ワード 64 ビットとしている．Brecv.numWords() は Brecv のワード数，popcount() は入力ワードに含まれる 1 になっているビット数を返す関数，“&” はビットワイズの AND 演算である．

図 **6** に図 2 の分割例の場合，頂点 0，1，2，4 がそれぞれメッセージ A，B，C，D を送信したときの例を示す．Mtmp のメッセージがない場所は「-」で表している．分かりやすくするため，メッセージ配列やビットマップを送信先ワーカまたは受信元ワーカごとに分けて表示しているが，実際には各ワーカが持っているこれらの配列は連続した 1 つの配列になっている．たとえば，ワーカ 2（W2）が受け取ったデータは単純に，Mrecv は (A,C)，Brecv はビット列 (1,0,0,0,1,0) である．

SendAll による最適化では，ビットマップ BhasEdge や入力エッジデータをあらかじめ構築しておく必要がある．グラフ処理では出力エッジだけで計算できるアプリケーションは多いので，その場合，本来必要なかった入力エッジを持つとメモリ使用量が増えることになる．

```
引数
localVertexes : このワーカに属する頂点集合
compute : ユーザ定義関数
Mrecv, Brecv : Alltoall で受け取ったデータ

def sendAllSuperstep(localVertexes, compute, Mrecv,
     Brecv) {
  val offset = makeOffset(Brecv);
  foreach(v in localVertexes) {
    val msgs = getMessage(Mrecv, Brecv, offset, v.
         inEdge());
    compute(new VertexContext(v, msgs));
  }
}

def getMessage(Mrecv, Brecv, offset, inEdge) {
  val msgs;
  foreach(origin in inEdge) {
    if(Brecv(origin)) {
      val wordOffset = origin / 64;
      val wordMask = (1 << (origin % 64)) - 1;
      val mesOffset = offset(wordOffset) + popcount(
           Brecv.word(wordOffset) & wordMask);
      msgs.add(Mrecv(mesOffset));
  }}
  return msgs;
}

def makeOffset(Brecv, offset) {
  val offset;
  offset(0) = 0;
  for(i in 0..Brecv.numWords()-1) {
    offset(i+1) = offset(i) + popcount(Brecv.word(i));
  }
  return offset;
}
```

図 5　各頂点に届いたメッセージを集めて compute() を呼び出す
Fig. 5　Looping over worker local vertices with received messages and the compute() closure.

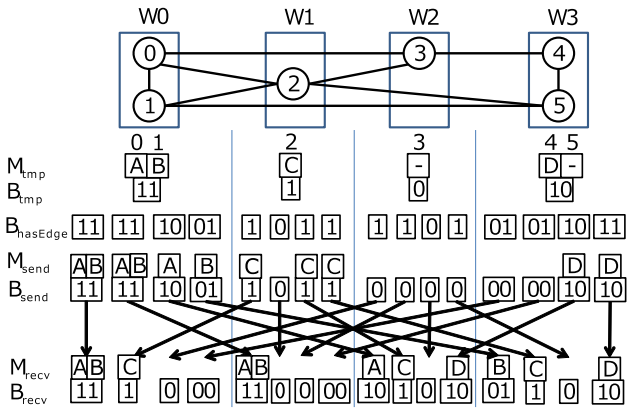


図 6　4 ワーカで頂点 0，1，2，4 がすべての隣接頂点にメッセージを送る場合の SendAll による処理例
Fig. 6　4 workers SendAll example when vertex 0, 1, 2, and 4 send a message to all of their neighbors.

各頂点に届いたメッセージを取得する処理はメモリへのランダムアクセスが発生し，計算負荷が比較的大きいので，必要なければ避けるのが望ましい．たとえば，図 **7** に示す幅優先探索の例では，探索で新たに訪問した頂点から，隣接頂点に頂点 ID を送るが，受け取る方は，すでに訪問済みの頂点の場合，メッセージを受け取る必要はなく，また，

```
(ctx :VertexContext[Double, Double, Long, Long]) => {
  ctx.voteToHalt();
  if(ctx.superstep() == 0) {
    ctx.setValue(-1L);
    if(ctx.id() != root) return ;
      ctx.setValue(root);
  }
  else {
    if(ctx.value() != -1L) return; // すでに訪問済み
    for(mgs in ctx) {
      ctx.setValue(mgs);
      break; // メッセージは 1 つ見ればよい
  }}
  ctx.sendMessageToAllNeighbors(ctx.id());
}
```

図 7　幅優先探索の実装例
Fig. 7　Breadth-first search example.

未訪問の頂点の場合でも，メッセージを 1 つ受け取れば訪問判定は可能で，2 つ目以降のメッセージを受け取る必要がない．図 5 に示す方法ではどのような場合でもすべての入力エッジを見ることになるが，XPregel では図 7 に示すようにメッセージの受け取りにイテレータパターンを用いていて，メッセージが送信されても，受け取り側でメッセージを受け取らなくてもよい場合，メッセージを集める処理を省略することができるようになっている．

3.3　計算量やデータ量の性質

SendAll におけるメッセージデータ量や計算量に関する考察を行う．ここで，V はグラフ全体の頂点数，E はグラフ全体のエッジ数，W はワーカ数，m は通常のメッセージ配送を行った場合の 1 スーパステップあたりのワーカ全体でのメッセージ数，d はメッセージ 1 つあたりのデータ量（バイト），α は Combiner を使った場合のメッセージ数削減率，β は SendAll を使った場合のメッセージ数削減率とする．Combiner を使った場合のメッセージ数は αm，SendAll を使った場合のメッセージ数は βm となる．

SendAll の 1 ワーカあたりの計算量オーダは，全頂点がメッセージを送信した場合，Bsend，Brecv などのビットマップ操作に $O(V)$，エッジが各ワーカに均等に分割されたとして，メッセージの取得処理に $O\left(\frac{E}{W}\right)$，その他の部分の計算量オーダはこれらと同じかそれ以下なので，合わせて，

$$O\left(V + \frac{E}{W}\right) \tag{1}$$

である．通常のメッセージ配送の場合の計算量オーダは，ソートがデータ量 n に対して $O(n)$ の基数ソートを使うとして，

$$O\left(\frac{E}{W}\right) \tag{2}$$

である．これらを比較すると，SendAll はグラフの全頂点 V が計算量オーダに入っていることから分かるとおり，ビットマップのサイズはグラフ全体の大きさに比例するので，ワーカ数を増やすと相対的に SendAll は不利になる．

```
(ctx :VertexContext[Double, Double, Double, Long]) => {
  var mindist :Double = (IsSource(ctx.realId())) ? 0 :
       INF;
  for(mes in ctx)
    if(mindist > mes)
      mindist = mes;
  if(mindist < ctx.value()) {
    ctx.setValue(mindist);
    for (val it = ctx.getOutEdgesIterator(); it.hasNext
         (); it.next()) {
      ctx.sendMessage(it.curId(), mindist + it.curValue
           ());
    }
  }
  ctx.voteToHalt();
}
```

図 8　単一始点最短経路の SendAll を使わない実装
Fig. 8　Single-source shortest paths without SendAll.

メッセージのデータ量は，通常のメッセージ配送を行った場合，メッセージ 1 つにつき，8 バイトの行き先頂点 ID と d バイトのメッセージ本体があるので，

$$(8 + d)m \tag{3}$$

Combiner を使った場合，

$$(8 + d)\alpha m \tag{4}$$

SendAll の場合，ビットマップの通信データ量が $\frac{VW}{8}$，メッセージのデータ量が $d\beta m$ なので，合計

$$\frac{VW}{8} + d\beta m \tag{5}$$

となる．よって，

$$\frac{VW}{8} + d\beta m < 8\alpha m + d\alpha m \tag{6}$$

の関係を満たせば Combiner よりも SendAll の方がデータ量は小さくなる．左辺の $\frac{VW}{8}$ はメッセージ数に関係なく，グラフ全体の頂点数と分割数で決まるのに対し，右辺も含めその他の部分はすべてメッセージ数に比例した値となるので，転送メッセージ数が少ない場合，左辺が大きくなり SendAll が不利になることが分かる．

3.4　適用可能範囲を広げるための拡張

エッジの重みを考慮するアルゴリズムの多くは，隣接頂点にメッセージは送るものの，接続しているエッジの重みに従って，メッセージの値が異なる場合がある．たとえば，単一始点最短経路問題（SSSP）は**図 8** に示すように，各隣接頂点に送るメッセージの値が異なっている．it.curValue() はエッジの重み（距離）である．

このような問題はメッセージ受信側で，エッジの重みを取得できるようにすることで，SendAll を使うことが可能である．**図 9** に SendAll を使う場合の単一始点最短経路問題の実装を示す．ctx.iterator() でメッセージイテレータを取り出しているが，このイテレータから入力エッジへのアクセスが可能となっている．it.edgeValue() は入力エッジの重み（距離）である．

```
(ctx :VertexContext[Double, Double, Double, Long]) => {
  var mindist :Double = (IsSource(ctx.realId())) ? 0 :
       INF;
  for(val it = ctx.iterator(); it.hasNext(); ) {
    val dist = it.next() + it.edgeValue();
    if(mindist > dist)
      mindist = dist;
  }
  if(mindist < ctx.value()) {
    ctx.setValue(mindist);
    ctx.sendMessageToAllNeighbors(mindist);
  }
  ctx.voteToHalt();
}
```

図 9　単一始点最短経路の SendAll を使う実装
Fig. 9　Single source shortest paths with SendAll.

このように，入力エッジからメッセージを再構築できる場合，出力エッジを使う処理を，入力エッジを使う処理に変更することで，多くのアルゴリズムに SendAll が適用可能である．

3.5　メッセージ数が少ない場合への最適化

SendAll では頂点に届いたメッセージを取得するのに図 5 にあるように，すべての入力エッジと対応するビットマップ Brecv を見る必要がある．これは送信されたメッセージ数とは関係なく必要な計算コストとなるので，メッセージ数が少ない場合，この固定コストが相対的に大きくなり，SendMessage() を使った通常のメッセージ配送の方が有利となる．そこで，メッセージ数が少ない場合は，SendAll によるメッセージ配送ではなく，通常のメッセージ配送に切り替えることで，処理の最適化を行った．

XPregel では，メッセージを送信する頂点数が全体の 2 割に満たない場合は，通常のメッセージ配送に切り替わるようになっている．

4.　性能評価

XPregel を使って，提案手法と既存手法を比較する．

4.1　評価環境

性能評価は，東京工業大学に設置されている TSUBAME2.5 を用いて行った．TSUBAME2.5 は各ノードに CPU Intel Xeon X5670 が 2 ソケットと，54 GB のメモリが搭載されている．ネットワークは Fat-tree の QDR Infiniband が dual-rail で構築されている．XPregel は X10 2.5.3 ベースの開発版 ScaleGraph を用いた．ScaleGraph の最新リリースバージョン 2.2 の XPregel にも SendAll を用いた最適化は実装されているが，本論文では改良されたバージョンで性能評価を行っている．

XPregel の実行は CPU1 ソケットあたり 1 ワーカプロセスで実行するため，各ノード 2 ワーカで実行した．たとえば，16 ノードでは，32 ワーカプロセスで実行している．グラフはエッジ数が頂点数に対して 16 倍の R-MAT グラフ

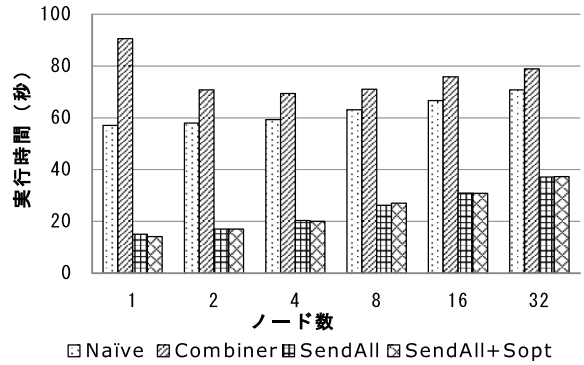


図 10　PageRank 30 イテレーション R-MAT グラフ Weak-Scaling（Scale 22～）
Fig. 10　PageRan 30 iteration, R-MAT graph, Weak-Scaling (Scale 22–).

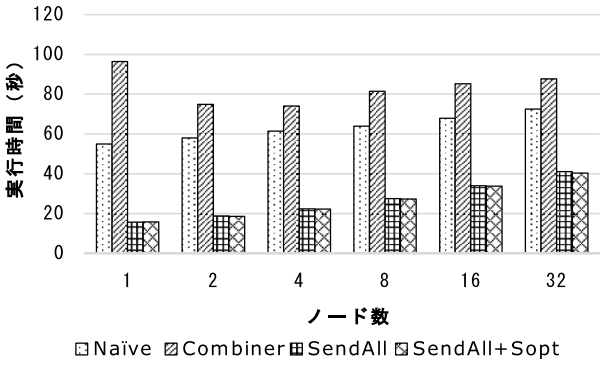


図 11　PageRank 30 イテレーションランダムグラフ Weak-Scaling（Scale 22～）
Fig. 11　PageRank 30 iteration, Random graph, Weak-Scaling (Scale 22–).

（A=0.45, B=0.15, C=0.15, D=0.30），および Erdős-Rényi ランダムグラフで行った．RMAT グラフ，ランダムグラフは大きさを Scale で表す．Scale N は頂点数が 2 の N 乗数のグラフである．ノード数に対するスケーラビリティ評価では，weak-scaling で評価を行った．Weak-scaling による評価では，1 ノードあたりの頂点数，エッジ数が同じになるように，1 ノードでの評価では Scale 22，2 ノードでの評価では Scale 23，4 ノードでは Scale 24 というように，ノード数に応じてグラフの頂点数，エッジ数を増やして評価を行った．

4.2　実行時間比較

図 10，**図 11**，**図 12**，**図 13** は，PageRank と幅優先探索を R-MAT グラフ，ランダムグラフを weak-scaling で計測した実行時間比較である．Naive は通常のメッセージ配送，Combiner は combiner() を使った場合，SendAll は少数メッセージに対する最適化なしの SendAll 適用時，SendAll+Sopt は少数メッセージに対する最適化ありの SendAll 適用時である．

SendAll+Sopt は Naive に比べて PageRank で 1.80 倍～

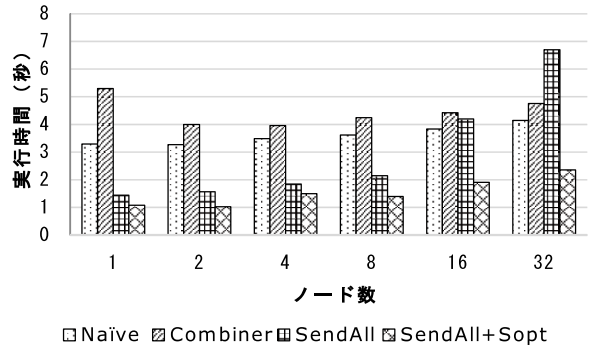


図 12　幅優先探索 R-MAT グラフ Weak-Scaling（Scale 22〜）
Fig. 12　Breadth-first search, R-MAT graph, Weak-Scaling (Scale 22–).

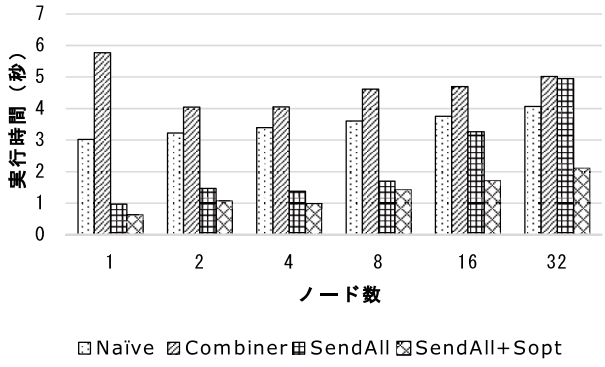


図 13　幅優先探索ランダムグラフ Weak-Scaling（Scale 22〜）
Fig. 13　Breadth-first search, random graph, Weak-Scaling (Scale 22–).

4.04 倍，幅優先探索で 1.76 倍〜4.76 倍，高速に計算できている．PageRank ではつねに全頂点がメッセージを送信するので少数メッセージへの最適化の効果はないが，幅優先探索では特に 16 ノード，32 ノードにおいて大きな効果が見られる．SendAll は 3.3 節での考察から，ワーカ数が増えると不利になることが分かっているが，それでも 64 ワーカ（32 ノード）で PageRank，幅優先探索ともに 1.8 倍前後の高速化がなされており，十分効果的であることが分かる．

4.3　転送データ量

図 14，**図 15**，**図 16**，**図 17** は幅優先探索計算時，および PageRank 計算時の転送メッセージ数，データ量である．少数メッセージへの最適化（Sopt）は行っていない．Combiner を使った場合と，SendAll を使った場合は，メッセージ数が通常のメッセージ配送の場合より小さくなっている．分割数を増やすと，同じノードに送られる重複メッセージが減ることからメッセージ削減が効きにくくなりメッセージ数は増加する．メッセージ数では，Combiner と SendAll であまり変わらないが，データ量は異なっている．これは，Combiner における各メッセージの行き先頂点 ID のデータ量と，SendAll でのビットマップのデータ量との差が表れている．つまり，Combiner の方が小さい場合は前者の方が小さい場合であり，SendAll の方が小さい場合は後者の方が小さい場合である．128 ノードでの

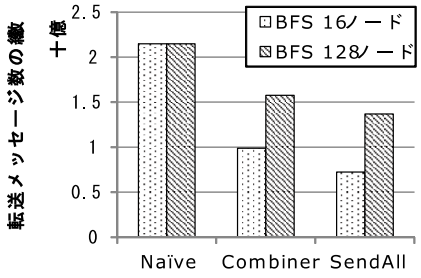


図 14　幅優先探索計算時の転送メッセージ数（Scale 26 RMAT）
Fig. 14　Number of messages transferred in Breadth-first Search (Scale 26 RMAT).

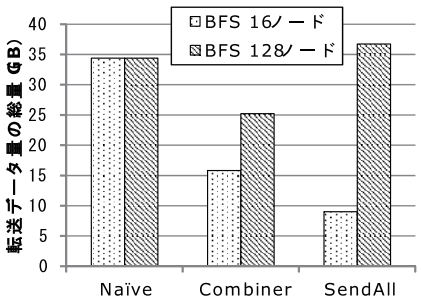


図 15　幅優先探索計算時の転送データ量（Scale 26 RMAT）
Fig. 15　Amount of data transferred in Breadth-first Search (Scale 26 RMAT).

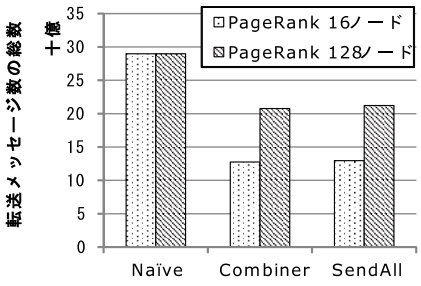


図 16　PageRank 計算時の転送メッセージ数（Scale 26 RMAT）
Fig. 16　Number of messages transferred in PageRank (Scale 26 RMAT).

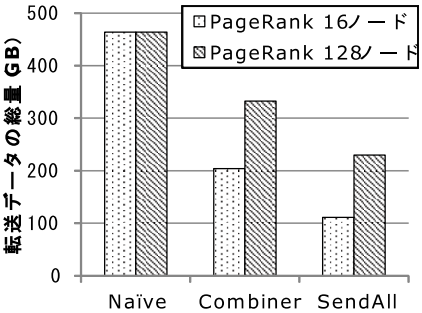


図 17　PageRank 計算時の転送データ量（Scale 26 RMAT）
Fig. 17　Amount of data transferred in PageRank (Scale 26 RMAT).

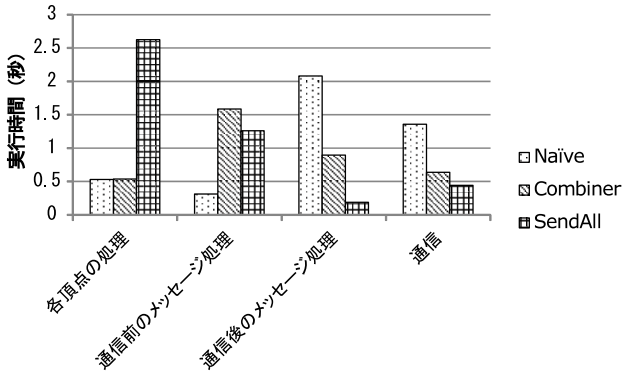


図 18　PageRank 計算時の実行時間内訳（Scale 26 RMAT，16 ノード）

Fig. 18　Breakdown of PageRank execution (Scale 26 RMAT, 16 nodes).

SendAll のデータ量が大きく増えているのは，式 (6) にあるとおり，SendAll はメッセージ数が少ない場合でも固定長の通信データがあり，その部分が大きく見えてしまっているためである．図の評価では SendAll の特性を見るため，少数メッセージへの最適化は適用していないが，16 ノード（32 ワーカ）の PageRank では，SendAll は Combiner に対して通信データ量が 45%削減，BFS でも 43%削減されている．

4.4　実行時間内訳

図 18 は，PageRank 計算時の実行時間内訳である．「各頂点の処理」では Naive，Combiner は compute() の呼び出しのみ，SendAll では図 5 にあるように compute() 呼び出しに加えて getMessage() によるメッセージの抽出を行っている．「通信前のメッセージ処理」では Naive，SendAll は通信データを作るためのメッセージデータのコピーなどを行っているが，Combiner ではそれに加えて combiner() によるメッセージの削減処理も行っている．「通信後のメッセージ処理」では Naive，Combiner はメッセージの行先頂点 ID でのソート，SendAll は図 5 の makeOffset() による処理を行っている．「通信」は MPI_Alltoall() および MPI_Alltoallv() の呼び出し時間である．通常のメッセージ配送では，通信後の受信メッセージのソートや通信に多くの時間がかかる．Combiner は通信前にメッセージ数を削減するので，通信や通信後のソートの時間は削減されているが，通信前に行うメッセージ数削減処理に多くの時間がかかっており，結局，全体の実行時間は減っていない．SendAll は各頂点の処理時に，getMessage() によるメッセージ抽出を行うので，Naive，Combiner に比べて各頂点の処理の実行時間が増えている．しかし，その他の処理は高速に行うことができるうえ，通信データ量が少ないので通信時間も小さくなっている．

5.　関連研究

Beamer らは幅優先探索の高速化手法として Direction optimization [10] を提案した．これは通常の探索方向である top-down 探索に加えて，状況によっては bottom-up 探索に切り替えることで見る必要のあるエッジ数を削減し幅優先探索を高速化するという手法である．Pregel において通常の SendMessage() による幅優先探索は top-down 探索となる．SendAll によるメッセージ配送とその計算の仕方は bottom-up 探索と似ており，幅広いアルゴリズムに適用可能な SendAll は Direction optimization を一般化したともいえる．

Shun らの Ligra [4] は Direction optimization [10] から着想を得て作られたグラフ処理系である．SendAll と同じく Direction optimization を一般化したともいえるが，Ligra は複数ノードでの分散処理に対応していない点，Pregel とは異なる新しいプログラミングモデルを提案しているという点で，SendAll とは異なる．

Salihoglu らによる GPS [5] は，グラフの動的再分割，次数の大きい頂点のエッジを複数ノードに分割する最適化手法 LALP，などに対応したグラフ処理系である．LALP はメッセージを受信側ワーカで分配するという点が SendAll と似ているが，SendAll はメッセージ配送処理全体に対する新しい処理方法の提案なのに対して，LALP は次数の大きい一部の頂点だけを対象にし，メッセージ配送処理の一部のみを置き換えるという点が，SendAll とは異なる．また，LALP は入力エッジのデータを取得することを想定していないため，SSSP などの処理を実装することは難しく，SendAll より制約が強い．

vertex-centric モデルを採用したグラフ処理系が多いなか，Tian ら [3] は graph-centric という新しいモデルを提案した．graph-centric は，アプリケーション実装に非常に強力な最適化を施すことが可能であり，vertex-centric に比べて 1〜2 桁の大幅な性能向上が可能であるが，graph-centric モデルでのグラフアルゴリズムの実装，および最適化は難易度が高いものとなっている．SendAll は一般に広く受け入れられている Pregel という簡単なプログラミングモデルに適用可能な最適化手法である．

GraphLab [11]，GraphX [12]，Giraph [13]，Giraphx [14] など，非常に多くのグラフ処理系が提案されているが，SendAll のような最適化手法は我々の知る限り提案されていない．

6.　まとめ

Pregel グラフ処理系の高速化手法として SendAll を提案した．SendAll はすべての隣接頂点に同じメッセージを送る場合に適用可能な最適化手法だが，API を少し拡張することにより多くのアルゴリズムに適用可能な手法である．

PageRank と幅優先探索の性能評価では，SendAll は適用しない場合と比較して，実行速度は PageRank で 1.80 倍～4.04 倍，幅優先探索で 1.76 倍～4.76 倍，通信データ量は半分程度に削減することに成功した．

謝辞　本研究は，JST-CREST の研究課題「ポストペタスケールシステムにおける超大規模グラフ最適化基盤」の支援による．

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, *Proc. 2010 ACM SIGMOD International Conference on Management of data, SIGMOD '10*, New York, NY, USA, ACM, pp.135–146 (online), DOI: 10.1145/1807167.1807184 (2010).

[2] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Comm. ACM*, Vol.51, No.1, pp.107–113 (online), DOI: 10.1145/1327452.1327492 (2008).

[3] Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S. and McPherson, J.: From "Think Like a Vertex" to "Think Like a Graph", *Proc. VLDB Endow.*, Vol.7, No.3, pp.193–204 (online), DOI: 10.14778/2732232.2732238 (2013).

[4] Shun, J. and Blelloch, G.E.: Ligra: A Lightweight Graph Processing Framework for Shared Memory, *SIGPLAN Not.*, Vol.48, No.8, pp.135–146 (online), DOI: 10.1145/2517327.2442530 (2013).

[5] Salihoglu, S. and Widom, J.: GPS: A Graph Processing System, *Proc. 25th International Conference on Scientific and Statistical Database Management, SSDBM*, New York, NY, USA, pp.22:1–22:12, ACM (online), DOI: 10.1145/2484838.2484843 (2013).

[6] Dayarathna, M., Houngkaew, C. and Suzumura, T.: Introducing ScaleGraph: An X10 Library for Billion Scale Graph Analytics, *Proc. 2012 ACM SIGPLAN X10 Workshop, X10 '12*, New York, NY, USA, pp.6:1–6:9, ACM (online), DOI: 10.1145/2246056.2246062 (2012).

[7] Suzumura, T., Ueno, K.C.H. and Ogata, H.: A High-Performance Library for Billion-Scale Graph Analytics, *IEEE BigData 2015* (2015).

[8] Valiant, L.G.: A Bridging Model for Parallel Computation, *Comm. ACM*, Vol.33, No.8, pp.103–111 (online), DOI: 10.1145/79173.79181 (1990).

[9] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, New York, NY, USA, pp.519–538, ACM (online), DOI: 10.1145/1094811.1094852 (2005).

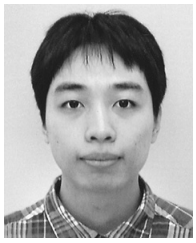
[10] Beamer, S., Asanović, K. and Patterson, D.: Direction-optimizing breadth-first search, *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, Los Alamitos, CA, USA, pp.12:1–12:10, IEEE Computer Society Press (2012) (online), available from ⟨http://dl.acm.org/citation.cfm?id=2388996.2389013⟩.

[11] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A. and Hellerstein, J.M.: Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud, *Proc. VLDB Endow.*, Vol.5, No.8, pp.716–727 (online), DOI: 10.14778/2212351.2212354 (2012).

[12] Xin, R.S., Gonzalez, J.E., Franklin, M.J. and Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark, *1st International Workshop on Graph Data Management Experiences and Systems, GRADES '13*, New York, NY, USA, pp.2:1–2:6, ACM (online), DOI: 10.1145/2484425.2484427 (2013).

[13] Giraph: available from ⟨http://incubator.apache.org/giraph/⟩.

[14] Tasci, S. and Demirbas, M.: Giraphx: Parallel Yet Serializable Large-scale Graph Processing, *Proc. 19th International Conference on Parallel Processing, Euro-Par'13*, Berlin, Heidelberg, pp.458–469, Springer-Verlag (2013).



上野 晃司　（学生会員）

1988 年生．2013 年東京工業大学大学院情報理工学研究科・計算工学専攻修士課程修了．現在，同専攻博士課程に在籍．大規模グラフ並列分散処理の研究に従事．開発した Graph500 ベンチマークの実装により「京」は Graph500 世界 1 位を獲得．2012 年度山下記念研究賞，大学生 OF THE YEAR 2014 特別賞受賞．

鈴村 豊太郎　（正会員）

1975 年生．2004 年東京工業大学大学院情報理工学研究科・数理計算科学専攻博士課程修了．同年より IBM Research 研究員．以来，ストリームコンピューティング，大規模グラフ処理基盤，大規模交通シミュレーション等ソフトウェアシステムに関する性能最適化の研究に従事．2015 年より米国 IBM トーマス・J・ワトソン研究所に所属．2009 年より 2013 年まで，東京工業大学大学院情報理工学研究科客員准教授を兼任，2014 年より同大学非常勤講師．2013 年よりアイルランド国立大学客員准教授を兼務．

松岡 聡　（正会員）

博士（理学）（東京大学 1993 年）．2001 年東京工業大学学術国際情報センター教授．高性能並列システムソフトウェア（GPU・省電力・高信頼・大規模データ処理）研究に従事．開発したスパコン TSUBAME2.0 は 2010 年 Top500 世界 4 位，2 期連続「世界一グリーンな運用スパコン」と認定．2013 年 TSUBAME-KFC が Green500 リストで世界一に．ACM/IEEE Supercomputing 含む多くの主要国際学会の委員長職等を歴任．情報処理学会坂井記念賞（1999 年），学術振興会賞（2006 年），ACM フェロー・ゴードンベル賞（2011 年），文部科学大臣表彰（2012 年），IEEE-CS シドニー・ファーンバック賞バック賞（2014 年）等受賞．