Vol.2016-HPC-153 No.10
2016/3/1

Bitmap Hybrid BFS の改良と「京」における性能評価

上野晃司[†1][†2]　鈴村豊太郎[†3]　丸山直也[†2]　松岡聡[†1]

概要: 近年，スパコンは大規模グラフ処理の潜在的な性能が認知され，スパコンの性能を活かす高速なグラフ処理の実現が求められている．我々は大規模分散メモリ環境で幅優先探索を高速に実行することができる Bitmap Hybrid BFS を開発し，「京」は Graph500 で 2014 年 6 月と 2015 年 6 月，11 月に 1 位を獲得した．2014 年 6 月と比較して 2015 年 6 月は 2 倍以上の性能となったが，本論文では，この性能向上の要因となった頂点 ID の並べ替えや通信の効率化などを提案する．特に提案手法による頂点 ID の並べ替えでは，15360 ノードでの評価でも 1.90 倍の性能向上が見られた．また，グラフの大規模分割に耐える疎行列表現についても，有効性について既存手法と詳細な比較を行った．

1. はじめに

グラフとは，頂点とエッジの集合である．近年，Web ページを頂点，ページ間のリンクをエッジとして見た Web グラフや，ソーシャルメディアサービスのユーザを頂点，ユーザ同士のつながりをエッジとみた，ソーシャルグラフなど，大規模なグラフデータがあり，これらのグラフを解析することへの関心が高まっている．また，脳の神経細胞のつながりをグラフ構造で表し，脳のシミュレーションに用いたり，たんぱく質の相互作用をグラフで表し，その性質を解析したりと，大規模グラフ処理は，生命科学分野でも必要とされつつある．

これらの大規模グラフ処理への関心の高まりを受けて，2010 年に Graph 500 [1] という新しいベンチマークが登場し，注目を集めている．Graph500 のランキングで用いられるのは大規模なグラフにおける幅優先探索の性能である．幅優先探索（Breadth-First Search, BFS）はグラフアルゴリズムの中でも最も基本的なアルゴリズムの１つであり，また，強連結成分分解や中心性解析などの複雑なアルゴリズムでも必要となる重要なアルゴリズムである．また，Graph500 ベンチマークで用いられる Kronecker グラフ[6]はスケールフリー性のある直径の短いグラフであり，これは Web グラフやソーシャルグラフ，生命科学分野で用いられるグラフなど，多くのグラフと同じような性質を持っている人口グラフである．Graph500 で用いられるグラフは現実世界でよく使われるグラフに近いと言える．

我々はこれまで，BFS の分散メモリ環境における高速化について研究[14][15][16]してきた．特に[16]では，本論文の提案する Bitmap Hybrid BFS のベースとなるアルゴリズムを提案している．このアルゴリズムは，ハイブリッド BFS[2]を，計算ノードが数千～数万あるような大規模なスーパーコンピュータ上で効率よく計算する手法である．本論文では，[16]の手法からの改良と，性能分析について述べる．本論文の貢献は以下の通りである．

1. 次数順 ID とオリジナル ID の併用や通信の効率化など，大規模分散メモリ環境で効率よく BFS を計算するためのデータ構造，アルゴリズムを提案
2. 従来手法との詳細な比較，解析
3. 「京」を用いた大規模性能評価とその結果

以降，２章ではハイブリッド BFS[2]のアルゴリズムとすでに提案されている隣接行列の２次元分割を使った手法[3]について説明し，３章で既存手法の問題提起，４章で提案手法のデータ構造やアルゴリズム，５章で「京」に沿った最適化，６章で性能評価を行い，６章で関連研究について述べ，７章で結論と今後の展望について述べる．

2. ハイブリッド BFS

2.1 基本アルゴリズム

通常の BFS の探索アルゴリズムは，図 1 にあるように，始点（source）から始めて，探索最前線（frontier）を外側に向かって広げるように探索する．この探索方向をトップダウンと呼ぶ．

これに対し，まだ，訪問していない頂点から，訪問済みの頂点が隣接頂点に含まれているかを検査するというのが，ボトムアップ方向の探索である．

Graph500 で使われる Kronecker グラフのような直径の短いグラフに対する BFS では，探索途中で探索方向（トップダウンとボトムアップ）を切り替えることにより，見る必要のあるエッジ数を削減し，探索を高速化することが可能である．トップダウン方向とボトムアップ方向を適切に使い分けて，見る必要のあるエッジ数がなるべく小さくなるように探索を進めるアルゴリズムをハイブリッド BFS[2]と呼ぶ．

†1 東京工業大学
Tokyo Institute of Technology
†2 理化学研究所
RIKEN
†3 IBM T.J. Watson Research Center

```
Function breadth-first-search (vertices, source)
1.  frontier ← {source}
2.  next ← {}
3.  parents ← [-1,-1,…,-1]
4.  while frontier ≠ {} do
5.  |  top-down-step (vertices, frontier, next, parents)
6.  |  frontier ← next
7.  |  next ← {}
8.  return parents

Function top-down-step (vertices, frontier, next, parents)
9.  for v ∈ frontier do
10. |  for n ∈ neighbors[v] do
11. |  |  if parents[n] = -1 then
12. |  |  |  parents[n] ← v
13. |  |  |  next ← next ∪ {n}
```

図 1 トップダウンアプローチによる BFS

```
Function bottom-up-step (vertices, frontier, next, parents)
1.  for v ∈ vertices do
2.  |  if parents[v] = -1 then
3.  |  |  for n ∈ neighbors[v] do
4.  |  |  |  if n ∈ frontier then
5.  |  |  |  |  parents[v] ← n
6.  |  |  |  |  next ← next ∪ {v}
```

図 2 ボトムアップアプローチの 1 ステップ

```
Function hybrid-bfs (vertices, source)
1.  frontier ← {source}
2.  next ← {}
3.  parents ← [-1,-1,…,-1]
4.  while frontier ≠ {} do
5.  |  if next-direction() = top-down then
6.  |  |  top-down-step (vertices, frontier, next, parents)
7.  |  else
8.  |  |  bottom-up-step (vertices, frontier, next, parents)
9.  |  frontier ← next
10. |  next ← {}
11. return parents
```

図 3 ハイブリッド BFS[2]

2.2 並列分散アルゴリズム

ハイブリッド BFS の分散メモリ環境における並列計算方法として，隣接行列の 2 次元分割を用いものが提案されている[3]．図 4 は隣接行列 A を R 行 C 列に 2 次元分割したものである．

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,C} \\ \vdots & \ddots & \vdots \\ A_{R,1} & \cdots & A_{R,C} \end{pmatrix}$$

図 4 隣接行列の 2 次元分割

計算ノードも仮想的に隣接行列と同じ R 行 C 列の 2 次元メッシュに配置され，分割された部分行列 $A_{i,j}$ は計算ノード $P(i,j)$ に割り当てられる．

2 次元分割された隣接行列を用いたトップダウンの探索アルゴリズム，ボトムアップの探索アルゴリズムをそれぞれ図 5, 6 に示す．

```
Function parallel-2D-top-down ( A , source)
1.  f ← {source}
2.  n ← {}
3.  π ← [-1,-1,…,-1]
4.  for all compute nodes P(i,j) in parallel do
5.  |  while f ≠ {} do
6.  |  |  transpose-vector( f_{i,j} )
7.  |  |  f_i = allgatherv( f_{i,j} , P(:,j) )
8.  |  |  t_{i,j} ← {}
9.  |  |  for u ∈ f_i do
10. |  |  |  for v ∈ A_{i,j}(:,u) do
11. |  |  |  |  t_{i,j} ← t_{i,j} ∪ {(u,v)}
12. |  |  w_{i,j} ← alltoallv( t_{i,j} , P(i,:) )
13. |  |  for (u,v) ∈ w_{i,j} do
14. |  |  |  if π_{i,j}(v) = -1 then
15. |  |  |  |  π_{i,j}(v) ← u
16. |  |  |  |  n_{i,j} ← n_{i,j} ∪ v
17. |  |  f ← n
18. |  |  n ← {}
19. return π
```

図 5 トップダウン 2 次元分割並列アルゴリズム

f, n, π はそれぞれ基本アルゴリズムにおける frontier, next, parent に対応する．アルゴリズム中の allgatherv, alltoallv は MPI の集団通信である．トップダウン探索では探索深さ 1 ステップを計算するのに transpose-vector, allgatherv, alltoallv を 1 回ずつ必要とする．2 次元分割 BFS[4]では，トップダウン探索における transpose-vector および allgatherv を Expand と呼び，その後の alltoallv を伴う計算を Fold と呼ぶ．ボトムアップ探索では，Expand 部分はトップダウンと同じであるが，Fold の計算がトップダウンとは異なっている．ボトムアップ探索では，探索深さ 1 ステップを計算するのに Fold 内で C サブステップを必要とする．Beamer ら[3]による手法では，高速化のため，アルゴリズム中の f，c，n，w を全て，1 頂点当たりを 1 ビットで表したビットマップを使って計算している．

ハイブリッド BFS は，図 4 のようにステップごとにトップダウン方向とボトムアップ方向から最適な方向を選んで計算する．図 5,6 は簡単のため，それぞれでアルゴリズムを独立に記述したが，2 次元分割並列ハイブリッド BFS は，ステップの区切りで探索方向切り替えられるようにし，図 4 のようにステップごとに探索方向を切り替えて計算できるようにしたものである．

```
Function parallel-2D-bottom-up ( A , source)
1.  f ← {source}
2.  c ← {source}
3.  n ← {}
4.  π ← [-1,-1,…,-1]
5.  for all compute nodes P(i, j) in parallel do
6.  |  while f ≠ {} do
7.  |  |  transpose-vector( f_{i,j} )
8.  |  |  f_i = allgatherv( f_{i,j} , P(:, j) )
9.  |  |  for s in 0…C-1 do
10. |  |  |  t_{i,j} ← {}
11. |  |  |  for u ∈ c_{i,j} do
12. |  |  |  |  for v ∈ A_{i,j}(u,:) do
13. |  |  |  |  |  if v ∈ f_i then
14. |  |  |  |  |  |  t_{i,j} ← t_{i,j} ∪ {(v,u)}
15. |  |  |  |  |  |  c_{i,j} ← c_{i,j} \u
16. |  |  |  |  |  |  break
17. |  |  |  w_{i,j} ← sendrecv( t_{i,j} , P(i, j+s) , P(i, j−s) )
18. |  |  |  for ( v,u ) ∈ w_{i,j} do
19. |  |  |  |  π_{i,j}(v) ← u
20. |  |  |  |  n_{i,j} ← n_{i,j} ∪ v
21. |  |  |  c_{i,j} ← sendrecv( c_{i,j} , P(i, j+1) , P(i, j−1) )
22. |  |  f ← n
23. |  |  n ← {}
24. return π
```

図 6 ボトムアップ 2 次元分割並列アルゴリズム

3. 大規模分散メモリ環境における問題

3.1 隣接行列データ構造の問題

隣接行列のデータ構造は，グラフ探索の計算量に大きく関わるので，非常に重要である．大規模にグラフを分割した場合，CSR (Compressed Sparse Row) などの単純な疎行列形式では必要なメモリ量が大きすぎて対応できない．この問題を解消するための DCSC[7]と Coarse index + Skip list [8]が提案されているが，について説明する．

3.1.1 CSR (Compressed Sparse Row)

CSR でグラフの隣接行列を表現する場合，エッジの行先の頂点番号を保持する配列 dst と，各頂点のエッジのオフセット row-starts の２本の配列で表現できる．したがって，CSR で必要になるメモリ量は，分割しない場合で考えると，頂点数 n，エッジ数 e の場合，dst の長さが e，row-starts の長さが n なので，

$$n+e \tag{1}$$

となる． $R\times C$ の２次元分割をした場合，エッジや頂点が均等に分割されたとすると，分割された部分行列の行数は n/R なので，部分隣接行列を CSR で表現するのに必要なメモリ量は

$$\frac{n}{R}+\frac{e}{RC} \tag{2}$$

となる．ここで，計算ノードあたりの頂点数を n'，グラフの平均次数を $\hat{d}$ とすると，

$$n'=\frac{n}{RC},\quad e=n\hat{d} \tag{3}$$

であるから(2)は以下のように表せる．

$$n'C+n'\hat{d} \tag{4}$$

C が大きくなるとメモリ使用量が増えるが，これは，row-starts のメモリ使用量が $n'C$ だからである．row-starts はエッジ配列への高速なアクセスをするために必要な配列ではあるが，C が大きい場合，row-starts が dst 配列よりも大きくなることがあり，問題である．

row-starts の圧縮方法として，DCSC[7]や Coarse index + Skip list [8]が提案されている．

3.1.2 DCSC

DCSC[7]は分散メモリ環境における行列積を効率よく行うため，行列を２次元分割して計算するために提案された手法である．非ゼロ要素が１つもない行のオフセットは持たないようにして row-starts を圧縮する．その代わり，目的の行にたどりつくために，JC 配列と AUX 配列という２つの配列を追加で持つ．この２つの配列により目的の行に高速にたどり着くことができるとされているが，AUX 配列から JC 配列をたどるときにループが必要になるので，計算コストが大きい．

3.1.3 Coarse index + Skip list

大規模分散メモリ環境における１次元分割による幅優先探索[8]のために提案された手法である．64 行分の非ゼロ要素データを１本のスキップリストにし，row-starts にはスキップリストへのポインタを持たせることで，row-starts を 64 分の 1 のサイズにする．１本のスキップリストに 64 行分のデータが連なっていて，１回スキップリストをたどる操作で複数行にアクセスすることも可能なので，アクセスする必要のある行が多い場合は効率がよい．ただし，64 行の 64 番目のみにアクセスする場合，最悪 63 回リストをたどらなければならないので，やはり計算コストが大きい．

3.1.4 その他の疎行列表現

row-starts を使わない疎行列表現[9]なら row-starts のメモリ使用量の問題を回避できる．しかし，これらの手法は，行列全体を読み取る必要がある場合には有効であるが，BFS で必要となる，各頂点のエッジを個別に取り出す必要がある場合，効率よく計算することができない．

3.2 通信データ量の問題

２次元分割ハイブリッド BFS は一定のノード数まではスケールするものの，計算ノード数が数千，数万規模の大規模環境になると，性能が頭打ちになる[3].

表 1 ボトムアップ探索の通信コスト[3]

操作	タイプ	1 ステップあたりの通信回数	1 探索あたりのデータ量（64bit ワード）
Transpose	1 対 1	O(1)	$s_b n/64$
Frontier Gather	allgather	O(1)	$s_b nR/64$
Parent Updates	1 対 1	O(C)	$2n$
Rotate Completed	1 対 1	O(C)	$s_b nC/64$

Graph500[1]の Kronecker[6]グラフのように直径の短いグラフにおけるハイブリッド BFS では，トップダウン探索よりボトムアップ探索の方に計算時間がかかることが知られている[3][5]が，表 1 は，f，c，n，w にビットマップを使った場合の，ボトムアップ探索における通信コスト[3]である．ここで，s_b はボトムアップで探索するステップ数，n はグラフの頂点数，R,C は隣接行列の分割数 $R \times C$ における R,C である．各操作は，Transpose が図 6 の 7 行目，Frontier Gather が 8 行目，Parent Updates が 17 行目，Rotate Completed が 21 行目に対応する．

表 1 から，Frontier Gather および Rotate Completed は，それぞれ隣接行列分割数の R,C に比例するコストがかかることが分かる．これが大規模環境で，ハイブリッド BFS の性能が頭打ちになる原因の 1 つである．また，アルゴリズム中 17,21 行目で他の計算ノードと通信しているが，この通信は，通信相手と同期する必要があり，さらに通信回数が C に比例するので，ここで発生する通信オーバーヘッドおよび，計算ノード間におけるロードインバランスも，大規模環境で性能が頭打ちになる原因となっている．大規模環境でのスケーラビリティを改善するには，これらの問題に対応しなければならない．

4. 提案手法

4.1 ビットマップを使った疎行列表現

グラフ大規模分割時の隣接行列データ構造の問題を解決するため，各頂点のエッジを効率よく取り出すことが可能であり，かつ，メモリ使用量を大幅に削減できる，ビットマップを使った疎行列表現を提案する．この手法は，CSR の row-starts を，エッジを 1 本以上持つ頂点のエッジ開始位置のみを保持するように圧縮し，各頂点についてエッジを 1 本以上持っているかどうかを 1 頂点当たり 1 ビットで表した bitmap で持つ．目的の頂点のエッジリストのエッジ配列における開始位置は，row-starts から読み出すが，CSR と違って row-starts は圧縮されているので，まず，目的の頂点の開始位置の row-starts 上での位置を計算する必要がある．この計算アルゴリズムを図 7 に示す．頂点 v が与えられたとき，頂点 v の row-starts 上での位置は，頂点 0 から v までのエッジを 1 本以上持っている頂点数と一致する．そして，これは bitmap の先頭から頂点 v に対応するビットまでの立っているビット数である．よって，これを計算するのだが，bitmap を始めから全て見ていくと効率が悪いので，あらかじめワード単位で，立っているビット数累計を計算しておき offset 配列に記憶しておく．offset 配列を使うことで，bitmap を高々 1 ワード見るだけで計算が可能となる．そして，この方法なら，どの頂点でも同じ計算量でエッジ開始位置を読み取ることができるようになる．図 7 に offset の計算アルゴリズムも示す. B は 1 ワードあたりのビット数，”<<”はビット単位のシフト演算，”&”はビット単位の and 演算，”mod”は剰余演算である．

```
Function make-offset (offset, bitmap)
1.  i ← 0
2.  offset[0] ← 0
3.  for each word w of bitmap
4.  |  offset[i+1] ← offset[i] + popcount(w)
5.  |  i ← i + 1

Function row-start-end (offset, bitmap, row-starts, v)
6.  w ← v / B
7.  b ← (1 << (v mod B))
8.  if (bitmap[w] & b) ≠ 0 then
9.  |  p ← offset[w] + popcount(bitmap[w] & (b-1))
10. |  return (row-starts[p], row-starts[p+1])
11. return (0, 0) // 頂点 v のエッジはない
```

図 7 ビットマップを使った疎行列表現における offset 配列作成アルゴリズム，および，頂点のエッジ範囲取得アルゴリズム

・エッジリスト

SRC	0	0	6	7
DST	4	5	3	1

・CSR

Row-starts	0	2	2	2	2	2	2	3	4
DST	4	5	3	1					

・Bitmapを使った疎行列表現

Offset	0	1	3					
Bitmap	1	0	0	0	0	0	1	1
Row-starts	0	2	3	4				
DST	4	5	3	1				

・DCSC（参考）

AUX	0	0	1	3
JC	0	6	7	
Row-starts	0	2	3	4
DST	4	5	3	1

図 8　ビットマップを使った疎行列表現の例

また，図 8 に頂点数 8，エッジ数 5 本の場合の例を示す．

表 2 は，CSR との比較である．ただし，p は分割された行列から 1 行取り出したときにその行にエッジが 1 本以上存在する確率である．実際のメモリ使用量の例として，64×32 分割（2048 分割）した 160 億頂点 2560 億枝の Graph500 で使用されるグラフの 1 分割あたりのデータ量も示す．

表 2 各疎行列表現のデータ量の理論値と例（64x32 分割した 160 億頂点 2560 億枝の Graph500 グラフの 1 分割あたりのデータ量）

データ	CSR		提案手法	
	理論値	例*	理論値	例*
offset	-	-	$n'C/64$	32MB
bitmap	-	-	$n'C/64$	32MB
row-starts	$n'C$	2048MB	$n'p$	190MB
dst	$n'\hat{d}$	1020MB	$n'\hat{d}$	1020MB
計	$n'(C+\hat{d})$	3068MB	$n'(C/32+p+\hat{d})$	1274MB

データ	DCSC		Coarse index + Skip list	
	理論値	例*	理論値	例*
AUX	$n'p$	190MB	-	-
JC	$n'p$	190MB	-	-
row-starts	$n'p$	190MB	$n'C/64$	32MB
dst または skip list	$n'\hat{d}$	1020MB	$n'\hat{d}+n'p$	1210MB
計	$n'(3p+\hat{d})$	1590MB	$n'(C/64+p+\hat{d})$	1242MB

4.2 頂点 ID 並べ替え

BFS の計算ではメモリへのランダムアクセスが発生するため，メモリアクセスの効率化が計算速度向上に寄与する．2.2 章で説明した通り，ハイブリッド BFS では多くの情報をビットマップで持っている．ビットマップは 1 頂点を 1 ビットで表したデータで，各頂点の情報を格納するビットの位置は頂点 ID から算出する．頂点によってビットマップへのアクセス頻度は異なるので，頻繁にアクセスする頂点の ID を小さい範囲にまとめることによって，頻繁にアクセスされるビットを局所化することが可能である．そのため，頂点 ID を次数順で付け替えることは BFS の高速化において重要である[14].

我々は[14]においてトップダウンのみの BFS における頂点 ID の並べ替えを提案した．この方法は必要なところだけ次数順 ID を使い，計算結果である BFS 木にはオリジナル ID を使うという画期的な手法であったが，そのままではハイブリッド BFS に適用できない．そこで，ハイブリッド BFS で使用可能な頂点 ID の並べ替え手法を提案する．

次数順 ID とは次数の大きい頂点から順に小さい ID を割り当てた頂点 ID である．次数順 ID は次のように計算する．

1. 各ノートに割り当てられた頂点の次数を計算
2. ノード内で次数の大きい順に頂点 ID の再割り当てを行う

頂点 ID から，その頂点の割り当てノードを計算する部分を変更せずに対応できるようにするため，ノード境界を超えた頂点 ID の移動や交換は行わない．そして，部分隣接行列は完全に次数順 ID で構築し，次数順 ID を用いて計算できるようにする．

ただし，この方法だと計算結果である BFS 木も次数順 ID で作られてしまうため，オリジナル ID に戻す必要がある．もし，この処理を BFS の最後にまとめて行う場合，オリジナル ID の情報を持っているのは頂点のオーナーノードだけなので，全対全通信が必要となる．大規模分散メモリ環境では，全対全通信は非常にコストの大きい通信である．

そこで，エッジの元，先，双方についてオリジナル ID を取得できるように，配列を 2 つ追加した．追加する 2 つの配列は図 9 の SRC(Orig)と DST(Orig)であり，共にオリジナル ID を保持している．BFS 木に書き込むときだけ，このオリジナル ID を使うことによって，全対全通信なしでも正しい出力を得られる．

頂点 ID 並べ替えの副次的な効果として，エッジが 1 本もない頂点を計算から除外してデータ量を小さくすることが可能である．BFS では，エッジが 1 本もない頂点は到達不可能なので，計算対象のグラフから除外しても計算結果になんら影響がない．

Offset	0	1	3					
Bitmap	1	0	0	0	0	0	1	1
SRC(Orig)	**2**	**0**	**1**					
Row-starts	0	2	3	4				
DST	2	3	0	1				
DST(Orig)	**4**	**5**	**3**	**1**				

図 9　ビットマップを使った疎行列表現に次数順 ID とオリジナル ID の両方を持った場合

4.3 ボトムアップ探索の通信効率化

図 6 のボトムアップ探索では，C 個サブステップを他の計算ノードと通信しながら進めているが，大規模環境では，サブステップの数が多くなるので，通信コストが大きい．この部分は以前の手法[16]では，Parent Updates をランダムな非同期通信で通信していたが，大規模に実行するとスケジューリングされていない通信は非常に効率が悪いことが分かった．

まず，以前の手法を説明する．図 6 の 17 行目の通信は，π (parents)を更新するリクエスト(Parent Updates)を送っているが，Parent Updates はいつ送ってもよく，他の処理を全て終えてから，Parent Updates をまとめて処理することも可

能である．そこで，通信可能となったデータから非同期通信で通信を開始していたのである．

しかし，全対全通信のような負荷の大きい通信は，適切にスケジューリングしないと効率が悪いので, 図 10 のように Parent Updates はまとめて alltoallv で通信することにした．

```
Function parallel-2D-bottom-up-opt ( A , source)
1.  f ← {source}
2.  c ← {source}
3.  n ← {}
4.  π ← [-1,-1,…,-1]
5.  for all compute nodes P(i, j) in parallel do
6.  |  while f ≠ {} do
7.  |  |  transpose-vector( f_{i,j} )
8.  |  |  f_i = allgatherv( f_{i,j} , P(:, j) )
9.  |  |  for s in 0…C-1 do
10. |  |  |  t_{i,j} ← {}
11. |  |  |  for u ∈ c_{i,j} do
12. |  |  |  |  for v ∈ A_{i,j}(u,:) do
13. |  |  |  |  |  if v ∈ f_i then
14. |  |  |  |  |  |  t_{i,j} ← t_{i,j} ∪ {(v,u)}
15. |  |  |  |  |  |  c_{i,j} ← c_{i,j} \ u
16. |  |  |  |  |  |  break
17. |  |  |  c_{i,j} ← sendrecv( c_{i,j} , P(i, j+1) , P(i, j−1) )
18. |  |  w_{i,j} ← alltoallv( t_{i,j} , P(i,:) )
19. |  |  for (v,u) ∈ w_{i,j} do
20. |  |  |  π_{i,j}(v) ← u
21. |  |  |  n_{i,j} ← n_{i,j} ∪ v
22. |  |  f ← n
23. |  |  n ← {}
24. return π
```

図 10 通信を効率化したボトムアップ探索

4.4 Top-Down ロードバランス

Top-Down では図 5 の 9〜11 行目のように，frontier の各頂点のエッジから $t_{i,j}$ を作る処理が必要である．これは，実装では，一時バッファにエッジの頂点ペアを入れて，alltoallv で通信する前に通信用メモリにコピーするという処理になっている．この処理をスレッド並列で行う場合，図 11 のように単純に frontier の頂点をスレッド数で分割して，この処理を行うと，各頂点のエッジ数には大きな不均衡があるので，一時バッファに頂点ペアを入れる処理で，大きなスレッド間ロードインバランスが発生する．

そこで，図 12 のように，行先ノードでスレッド分割して処理を行えるようにした．この方法では，まずはエッジ範囲を抽出して，その後，一時バッファを介さずに直接通信メモリへのコピーするようになっている. 図中 owner(v)は，v の担当ノードを返す関数，edge-range($A_{i,j}(:,u), k$)は，エッジリスト $A_{i,j}(:,u)$ の，担当ノード k の範囲を 2 分探索で検索して返す関数である．エッジリストは行先頂点 ID 順に並んでいるので，担当ノードごとに連続になっている．

図 12 の方法では，一時バッファへの無駄なコピーも省くことができ，その点は有利だが，エッジが少ない頂点を処理する場合，エッジ範囲データ（図中 $r_{i,j,k}$ ）の方が大きくなる可能性があり，効率が悪い場合がある．そこで，頂点のエッジ数により，単純な方法と同じように一時バッファに入れてから通信用メモリにコピーする方法と，範囲データを使って直接コピーする方法の２つを動的に選択して処理するハイブリッド手法を提案する．

```
Function top-down-sender-naive ( A_{i,j} , f_i )
1.  for u ∈ f_i in parallel do
2.  |  for v ∈ A_{i,j}(:,u) do
3.  |  |  k ← owner( v )
4.  |  |  t_{i,j,k} ← t_{i,j,k} ∪ {(u,v)}
```

図 11 トップダウンの単純なスレッド並列

```
Function top-down-sender-load-balanced ( A_{i,j} , f_i )
1.  for u ∈ f_i in parallel do
2.  |  for k ∈ P(i,:) do
3.  |  |  (v0,v1) ← edge-range( A_{i,j}(:,u) , k )
4.  |  |  r_{i,j,k} ← r_{i,j,k} ∪ {(u,v0,v1)}
5.  for k ∈ P(i,:) in parallel do
6.  |  for (u,v0,v1) ∈ r_{i,j,k} do
7.  |  |  for v ∈ A_{i,j}(v0:v1,u) do
8.  |  |  |  t_{i,j,k} ← t_{i,j,k} ∪ {(u,v)}
```

図 12 トップダウンのロードバランスを考慮したスレッド並列

5. マシンアーキテクチャに沿った通信の最適化

「京」のノード間通信接続のトポロジは 6 次元トーラス・メッシュとなっている．Bitmap Hybrid BFS は 2 次元分割を用いた手法なので，R 行 C 列のノードの２次元配置において，６次元の物理トポロジのうち R 軸に３次元，C 軸に３次元割り当てることで，２次元配置における隣接ノードを，物理配置においても隣接させることが可能である．また，「京」全体を使った実行の場合，「京」の 6 次元の軸 x,y,z,a,b,c のうち，R 軸に y,z 軸，C 軸に x,a,b,c 軸を割り当てることで 288×288 の２次元ノード配置を作ることができる．

6. 性能評価

本論文の提案手法を「京」を使って Graph500 ベンチマークで用いられるグラフを対象とした性能評価を行った．

6.1 Graph500 ベンチマーク

Graph500 ベンチマークでは，１つの巨大なグラフを，スパコンの１つのシステム全体で計算して，その処理の速さをスパコンのグラフ処理性能として，ランキングに利用する．グラフ処理の速度は，単位時間に処理できたエッジ数 TEPS (Traversed Edges Per Second)で表現される．グラフはパラメータが A=0.57, B=0.19, C=0.19, D=0.05 の Kronecker

グラフである．グラフのサイズは，グラフの頂点数=2^{SCALE}であるような Scale 値で表し，エッジ数は頂点数の 16 倍である．

6.2 「京」

「京」は神戸の AICS に設置されているスーパーコンピュータである．各計算ノードは SPARC64 VIIIfx CPU 8 コアを 1 つと 16GB のメモリを備えている．各計算ノードは 6 次元メッシュ/トーラスで接続され，帯域は各計算ノード間の接続 1 本あたり 5GB/s×2（双方向）である．

6.3 提案手法の効果

各種提案手法の効果を「京」15360 ノードまで使って検証する．「京」60 ノードあたり Scale 29 相当のグラフサイズ（頂点数 5 億 3687 万頂点，エッジ数 85 億 8993 万）のウィークスケーリングで BFS を実行した．つまり，最大ノード数 15360 ノードではグラフサイズ Scale 37 で実行している．始点をランダムに重複なしで選択し BFS を 300 回実行し，300 回中の中央値を性能とした．

まず，ビットマップを使った疎行列表現を従来手法である DCSC[7], Coarse index + Skip list[8]と比較する．図 13 はウィークスケーリングによる評価，図 14 は 15360 ノードで実行したときのグラフ読み取り部分にかかった時間の比較である．図から，提案手法は DCSC や Coarse index + Skip list よりも高速に処理できることが分かる．特に，グラフ読み取り部分だけに着目すると，提案手法は DCSC の 5.5 倍，Coarse index + Skip list の 3.0 倍の速度で処理している．

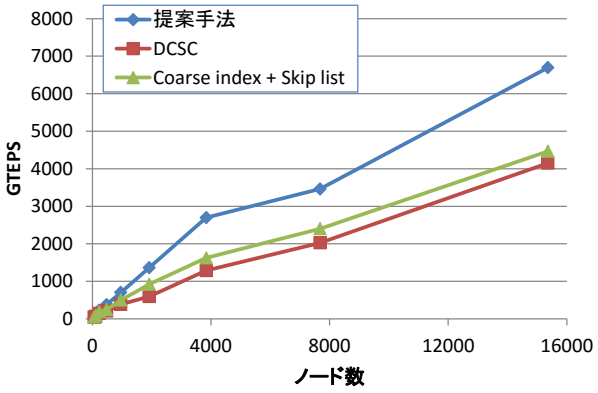

図 13 ビットマップを使った疎行列表現と従来手法の比較（「京」ウィークスケーリングによる評価）

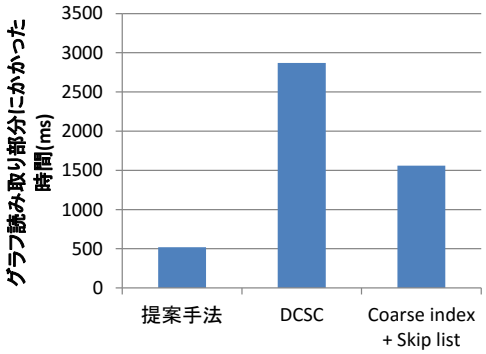

図 14 ビットマップを使った疎行列表現と従来手法の比較（15360 ノードで Scale37 を実行）

次に，頂点 ID の並べ替えに関して，図 15 に提案手法である次数順 ID とオリジナル ID を併用する手法と，他の手法の比較を示す．他の手法として，「最後にオリジナル ID に戻す」は，オリジナル ID を使わないで BFS 木を作成し，最後に，全対全通信によって BFS 木をオリジナル ID に戻すという手法，「リオーダーなし」は頂点 ID 並べ替えを一切行わないこと，「次数ゼロの頂点だけを取り除く」は，ID の付け替えは行うが，次数ゼロの頂点の ID を後ろに持っていくだけで，他の頂点は次数順にはしないで元の順番を維持するという手法である．頂点 ID の並べ替えは，メモリアクセスの効率化と次数ゼロの頂点が除外されることによるデータ量の削減の 2 つの効果があり，この 2 つの効果を独立に検証できるようにするため，次数ゼロの頂点だけを取り除いた場合の評価も行った．

図 15 から，最後にオリジナル ID に戻す手法は，予想通り全対全通信のコストが大きく，リオーダーなしよりも性能が低くなってしまった．次数ゼロの頂点を取り除くだけで，比較的大きな性能向上がなされているのが分かるが，これは Graph500 ベンチマークのグラフの特性から，15360 ノードで実行した Scale 37 では半数以上の頂点の次数がゼロであることに起因する．頂点 ID を次数順にすることにより，さらに性能が向上しているのが分かる．

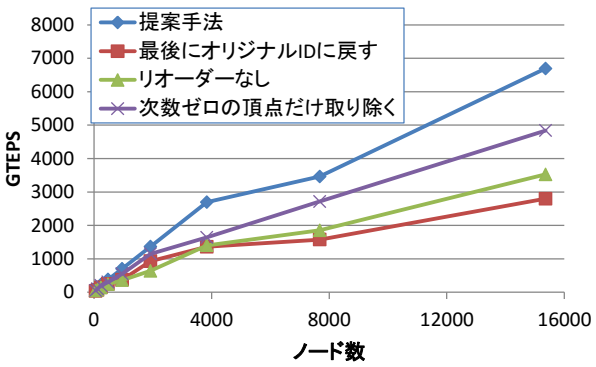

図 15 頂点 ID 並べ替えの各種比較（提案手法は，次数順 ID とオリジナル ID を併用した計算）

次に，トップダウン探索におけるロードバランスのための提案手法の効果を見る．図 16 は提案手法と他の手法の比較である．他の手法は，「サイズを数えてからコピー」とはすべて図 11 のアルゴリズムで処理した場合，「すべて一時バッファに入れてからコピーとは，すべて図 12 のアルゴリズムで処理した場合，提案手法はそのハイブリッド手法である．切り替えは，部分隣接行列における頂点のエッジリストの長さが 1000 以上かどうかで行った．提案手法は効果が不安定，かつ，あまり大きな差ではないが，確かに性能が向上していることが分かる．

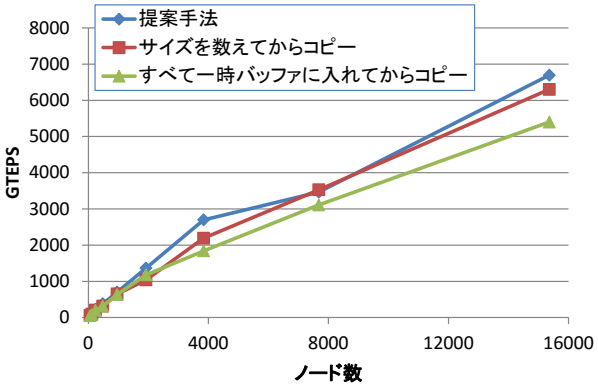

図 16 トップダウン探索におけるロードバランス

図 17 にこれらの最適化を積み上げた比較を示す．Naïve 版は，DCSC を使い，頂点リオーダーなしで，トップダウン処理はすべて図 11 のアルゴリズムを適用した場合である．提案手法は Naïve 版と比較して 15360 ノードで 3.19 倍の性能となっている．

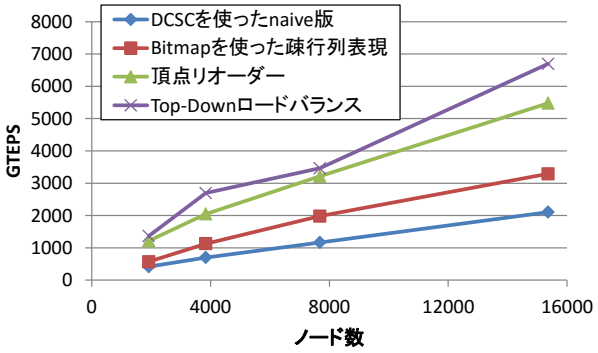

図 17 各高速化手法の累積積み上げ効果

図 18 に提案手法をウィークスケーリングで実行時の 1 ノードあたりの性能を示す．また，図 19 に 60 ノード，15360 ノードで実行時の BFS 1 回あたりの実行時間内訳を示す．問題サイズは図 18 の同ノード数実行時と同じである．図 18 から 1 ノードあたりの性能はノード数が大きくなると下がっているが，図 19 の実行時間内訳から通信とプロセス間同期待ちの増加がその原因であることが分かる．

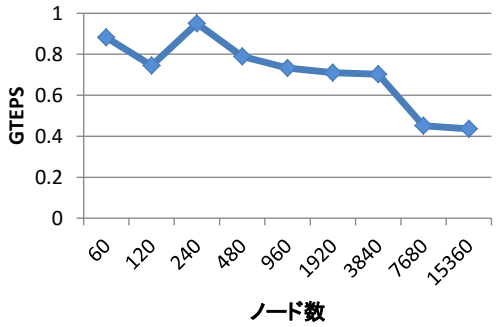

図 18 1 ノード当たりの性能比較

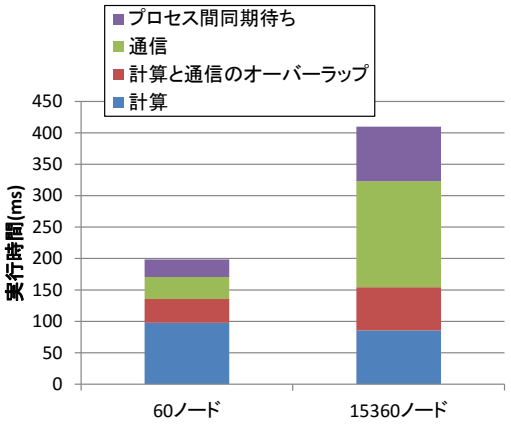

図 19 実行時間内訳

6.4 「京」全ノード使用時の性能

「京」の全 82944 ノードを使用して，提案手法による BFS 実装した Graph500 ベンチマークを実行し，38621.4 GTEPS を達成し，2015 年 6 月の Graph500 リストで「京」は 1 位を獲得している．

7. 関連研究

大規模分散メモリ環境における BFS の研究として，まず，Yoo らによる 2 次元分割 BFS[4]がある．彼らは，シンプルなトップダウンアルゴリズムによる 2 次元分割 BFS を提案し，BlueGene/L 32768 ノードでの性能評価を行った．

Buluç ら[10]は，Hopper（6392 ノード）や Franklin（9660 ノード）といったスーパーコンピュータを用いて BFS の 1 次元分割と 2 次元分割の性能比較を行った．Satish ら[11]は，Intel CPU と Infiniband ネットワークのスーパーコンピュータにおける効率の良い分散並列 BFS を提案した．Checconi ら[12]は，スーパーコンピュータ BlueGene における Wave を使った効率の良い分散並列 BFS を提案した．しかし，これらの手法は全てトップダウン探索のみを使った

手法であり，直径の比較的短いグラフに対して有効なハイブリッド BFS は使っていない．

ハイブリッド BFS を使った分散メモリ環境での BFS の研究は，本論文の手法のベースとなった，Beamer ら[3]による手法や，Checconi ら[13]による 1 次元分割の手法がある．Checconi らによる手法は，1 次元分割のシンプルなアルゴリズムをベースに次数の大きい頂点を全計算ノードで共有するロードバランスといった最適化手法を取り入れた画期的な手法である．BlueGene/Q 65536 ノードを使った性能評価では，16,599GTEPS の性能を達成している．

8. 結論と今後の展望

グラフの大規模分割に耐える疎行列表現として，ビットマップを使った疎行列表現を提案していたが，DCSC[7]や Coarse index + Skip list[8]などの既存手法と比較して，提案手法は優れていることが確かめられた．また，次数順 ID とオリジナル ID の併用による頂点 ID 並べ替えは，少ないオーバーヘッドで，計算の高速化を達成した．これらの高速化により, Bitmap Hybrid BFS は Naïve 版と比較して 15360 ノードで 3.19 倍の性能となった．また，「京」全ノードを使った Graph500 ベンチマークによる性能評価では，38621.4 GTEPS を達成し，2015 年 6 月の Graph500 リストで「京」は 1 位を獲得している．

今後，詳細なパフォーマンスモデリングを行い，ハードウェアの性能と，Graph500 ベンチマークの性能がどのような関係にあるのかを明らかにしていきたい．

謝辞 本研究は JST CREST の支援により行われたものである．

参考文献

1) Graph500 : http://www.graph500.org/
2) Scott Beamer, Krste Asanović and David Patterson. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12).
3) Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW '13).
4) Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05). IEEE Computer Society, Washington, DC, USA.
5) Yuichiro Yasui, Katsuki Fujisawa and Yukinori Sato. Fast and Energy-efficient Breadth-First Search on a Single NUMA System. 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014.
6) J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication, in Conf. on Principles and Practice of Knowledge Discovery in Databases, 2005.
7) Aydın Buluç and John R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. Parallel and Distributed Processing Symposium 2008 (IPDPS'08).
8) Fabio Checconi, et. al. Traversing Trillions of Edges in Real-time: Graph Exploration on Large-scale Parallel Machines. Parallel and Distributed Processing Symposium 2014 (IPDPS'14).
9) Eurípides Montagne and Anand Ekambaram. An optimal storage format for sparse matrices. Journal Information Processing Letters Volume 90 Issue 2, 30 April 2004 Pages 87 - 92.
10) Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). ACM, New York, NY, USA, Article 65 , 12 pages. DOI=10.1145/2063384.2063471 http://doi.acm.org/10.1145/2063384.2063471
11) Satish, Nadathur and Kim, Changkyu and Chhugani, Jatin and Dubey, Pradeep. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. SC '12, 2012.
12) Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, Yogish Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. SC '12, 2012.
13) Fabio Checconi, et. al. Traversing Trillions of Edges in Real-time: Graph Exploration on Large-scale Parallel Machines. IPDPS'14.
14) Koji Ueno and Toyotaro Suzumura "Highly Scalable Graph Search for the Graph500 Benchmark" HPDC 2012 (The 21st International ACM Symposium on High-Performance Parallel and Distributed Computing) 2012/6, Delft, Netherlands.
15) Koji Ueno and Toyotaro Suzumura, "Parallel Distributed Breadth First Search on GPU", HiPC 2013 (IEEE International Conference on High Performance Computing), India, 2013/12.
16) 上野 晃司，鈴村 豊太郎，丸山直也，松岡聡，「大規模分散メモリ環境におけるハイブリッド BFS の最適化」, HPC 研究会, 2014/9