Vol.2014-HPC-146 No.21
2014/10/3

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

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，Web グラフやソーシャルグラフなど大規模なグラフデータが多くあり，大規模グラフ解析への関心が高まっている．本論文では，比較的直径の短いグラフで有効な幅優先探索（BFS）アルゴリズムであるハイブリッド BFS 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を提案する．ビットマップを使った疎行列表現や，頂点濃度に応じたデータ構造選択，ボトムアップ探索の並列性を上げることによる効率化を行い，数万ノード規模でのスケーラビリティを得られた．「京」を使った性能評価では，65,536 ノードで 17,997GTEPS の性能を達成し，2014 年 6 月の Graph500 ランキングにおいて「京」は 1 位を獲得した．

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，2011 年に Graph 500 [1] という新しいベンチマーク登場し，注目を集めている．従来，スーパーコンピュータは物理シミュレーションなどの数値計算に，主に使われてきたが，大規模グラフ処理は，数値計算とは性質が異なる処理であり，スパコンのグラフ処理性能でランキングされる Graph500 は，スパコンの性能を知る有用な指標である．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グラフは Kronecker グラフ[6]である．このグラフはスケールフリー性のある直径の短いグラフである．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，多くのグラフはスケールフリー性があり，必然的にグラフの直径比較的短くなっており，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]してきた．本論文では，比較的直径の短いグラフで有効な BFS アルゴリズム，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を提案する．本論文の貢献は以下の通りである．

1. 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. トーラスネットワーク, NUMA アーキテクチャなどスーパーコンピュータの特性に合わせた最適化手法の提案
3. TSUBAME2.5 と「京」の全形を用いた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章で提案手法のデータ構造やアルゴリズム，４章で３次元トーラスネットワーク，NUMA アーキテクチャなどスーパーコンピュータの特性に合わせた最適化手法を説明する．５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．

†1 東京工業大学
Tokyo Institute of Technology
†2 理化学研究所
RIKEN
†3 ユニバーシティ・カレッジ・ダブリン
University College Dublin

```
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

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．

```
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．

```
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

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．

2.3 大規模分散メモリ環境における問題

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．

3. 提案手法

3.1 データ構造

3.1.1 従来手法の問題

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．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 の場合で，

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

となる．$R\times C$ の２次元分割をした場合，分割された部分行列を 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，(2)は以下のように表せる．

$$n'(C+\hat{d}) \tag{3}$$

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]やスキップリスト[8]が提案されている．また，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．

3.1.2 ビットマップを使った疎行列表現

そこで，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．頂点 v が与えられたとき，頂点 v 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, bitmap を使って位置を計算する．つまり, row-starts には，bitmap から頂点 v までに，エッジを 1 本以上持つ頂点の数が, row-starts 上での頂点 v のエッジ開始位置なので，これを計算する．このとき，bitmap を始めから見ていくと効率が悪いので，あらかじめワード単位で計算しておき offset 配列に記憶しておくことで，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．offset の計算アルゴリズムと，頂点 v のエッジ範囲を計算するアルゴリズムを図 7 に示す．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	0	6	7
dst	4	5	7	3	1

・CSR

row-starts	0	3	3	3	3	3	3	4	5
dst	4	5	7	3	1				

・ビットマップを使ったCSR

offset	0	3							
bitmap	1	0	0	0	0	0	1	1	
row-starts	0	2	3	4					
dst	4	5	7	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 ビットマップを使った CSR 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		ビットマップを使った CSR	
	理論値	例*	理論値	例*
offset	-	-	$n'C/64$	32MB
bitmap	-	-	$n'C/64$	32MB
row-starts	$n'C$	2048MB	n'	190MB
dst	$n'\hat{d}$	1020MB	$n'\hat{d}$	1020MB
計	$n'(C+\hat{d})$	3068MB	$n'(1/32+p+\hat{d})$	1274MB

3.2 頂点濃度に応じたデータ構造選択

Beamer ら[3]によるハイブリッド BFS では，トップダウンの通信では疎ベクトルを用い，ボトムアップの通信ではビットマップを用いる．ボトムアップでは通信する頂点数が多いので，通信がさほど問題とならない小規模環境（計算ノード数 1000 程度まで）においては，ビットマップで通信しても大きな問題とはなっていなかった．しかし，大規模環境ではビットマップで通信するデータは，R や C に比例して通信データサイズが増えてしまい，これがボトルネックになってしまう．

そこで，本論文では，データの頂点濃度に応じてビット

マップと疎ベクトルから最適なデータ構造を選択して通信する手法を提案する．ボトムアップでボトルネックとなる通信データは，表 1 において，通信データ量が R や C に比例する，Frontier Gather と Rotate Completed である．疎ベクトルで通信した場合，Frontier Gather のサイズは，frontier の頂点数に比例し，Rotate Completed のサイズは，未訪問の頂点数に比例する．疎ベクトルでは各頂点を 64bit[a]で表すとすると，頂点数が全体の頂点数の 64 分の 1 を超えた場合，ビットマップのほうがデータサイズは小さくなり，下回った場合は，疎ベクトルの方がデータサイズは小さくなる．図 9 に Scale40 の Graph500 グラフにおけるレベルごとの frontier の頂点数と未訪問頂点数の例を示す．この例の場合，レベル 4 から 6 をボトムアップ，そのほかのレベルトップダウンで処理している．しかし，レベル 6 の未訪問の頂点数と，レベル 4 の frontier の頂点数は，全体の頂点数の 64 分の 1 を下回っているので，疎ベクトルで通信したほうがデータサイズは小さくなることが分かる．そこで，この 2 つは疎ベクトルで通信し，ボトムアップの他の部分ではビットマップを使って通信したほうがよいことが分かる．

Frontier Gather に疎ベクトルを用いた場合，ボトムアップ探索を実行するのにはビットマップに展開する必要があるので，各計算ノードは疎ベクトルを受信した後，ビットマップに展開する．Rotate Completed に疎ベクトルを使った場合，前節で述べたビットマップを使った CSR を用いることで，疎ベクトルのままでも効率よく処理することが可能なため，ビットマップには展開せず，疎ベクトルのまま送受信すればよい．

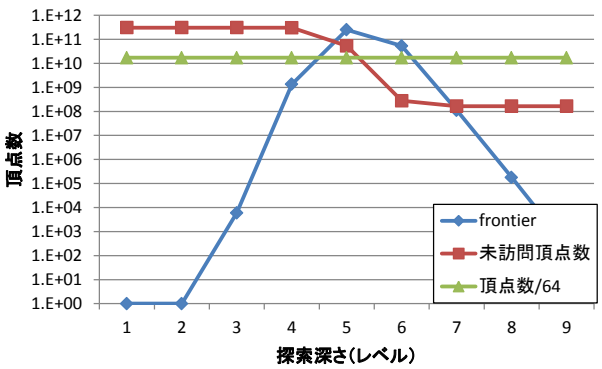

図 9　1 兆頂点 16 兆エッジの Graph500 グラフにおけるレベルごとの frontier の頂点数と未訪問頂点数の例

3.3　ボトムアップ探索の効率化

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同期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，同期コストが大きくなってします．サブステップを順番に進めるのではなく，順番に関係なく受信したデータから処理を進めるという手法を提案する．

3.3.1 Parent Updates 通信

まず，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，以下の 2 種類を実装し比較した．

1. サブステップの処理と同時に Parent Updates も通信するが，実際の更新処理サブステップの処理が全て終えてから行う．
2. Parent Updates の通信も全サブステップを終えてからまとめて行う．
3. サブステップの処理と同時に Parent Updates も通信し，実際の更新処理も同時に行う

3 は，更新処理を計算ノード内の各スレッドに均等に分配することが難しく，ノード内のロードインバランスによって性能が得られないことが分かった．1 と 2 については，サブステップ数が少ない場合は，1 の方法が良く，サブステップ数が多い場合は，2 の方法が良いことが分かった．これは，サブステップ数が少ない場合は，図 6 の 21 行目で行っている Rotate Completed よりも Parent Updates の方がデータ量は多く，かつサブステップ内の計算量が多いので，Rotate Completed と Parent Updates を同時に通信したほうが，計算と通信の両方にバランス良く負荷をかけることができる．それに対して，サブステップ数が多い場合は，Rotate Completed の通信データ量が多く，計算量は相対的に小さくなるため，Parent Updates を同時に通信すると通信が混雑，あるいは，通信のロードインバランスが発生するため，別に通信したほうが効率よく処理できると考えることができる．

3.3.2 Rotate Completed 通信

Rotate Completed は，通信と計算をオーバーラップさせるため，サブステップ数を従来の C 個から 2 または 4 倍の 2C 個または 4C 個に増やし，2~4 個を同時に通信または計算できるようにした．また，この時，通信するデータの方にラベルを付けることで，処理する順番は，ループの順番とは関係なく，受信した順に処理できるようにすることで，無駄な待ち時間を減らしている．

3.4　分割方法の改善

Beamer らによる 2 次元分割[3]では単純な分割を用いているので，ステップ毎に transpose-vector による通信が必要となる．Yoo らによる 2 次元分割 BFS[4]は，ブロックサイクリックな分割を用いているが，この手法の利点は

a 余分なビットを切り落とせば 32bit にすることも可能であり，実際にベンチマークなどにおいては 32bit で実装している．

transpose-vector による通信が必要ないことである．分割方法は Beamer らの手法に比べて少し複雑にはなるが，それによる計算オーバーヘッドはなく，ハイブリッド BFS にも適用可能なので，本論文では，Yoo らによる２次元分割を用いた．

$$\begin{bmatrix} A_{1,1}^{(1)} & A_{1,2}^{(1)} & \cdots & A_{1,C}^{(1)} \\ A_{2,1}^{(1)} & A_{2,2}^{(1)} & \cdots & A_{2,C}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ A_{R,1}^{(1)} & A_{R,2}^{(1)} & \cdots & A_{R,C}^{(1)} \\ A_{1,1}^{(2)} & A_{1,2}^{(2)} & \cdots & A_{1,C}^{(2)} \\ A_{2,1}^{(2)} & A_{2,2}^{(2)} & \cdots & A_{2,C}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ A_{R,1}^{(2)} & A_{R,2}^{(2)} & \cdots & A_{R,C}^{(2)} \\ & & & \\ & & & \\ & & & \\ A_{1,1}^{(C)} & A_{1,2}^{(C)} & \cdots & A_{1,C}^{(C)} \\ A_{2,1}^{(C)} & A_{2,2}^{(C)} & \cdots & A_{2,C}^{(C)} \\ \vdots & \vdots & \ddots & \vdots \\ A_{R,1}^{(C)} & A_{R,2}^{(C)} & \cdots & A_{R,C}^{(C)} \end{bmatrix}$$

図 10 Yoo らによる２次元分割[4]

4. マシンアーキテクチャに沿った最適化

本論文の提案手法をスーパーコンピュータ上で実装する場合，MPI と OpenMP のハイブリッド並列で実装可能である．その場合の，各スーパーコンピュータのマシンアーキテクチャに沿った最適化手法を提案する．

4.1 共有メモリを使った通信データ量の削減

東工大の TSUBAME2.5 のようにマルチソケット CPU を備えたスーパーコンピュータは多い．近年のマルチソケット CPU はほとんどが NUMA アーキテクチャなので，CPU ごとにアクセスするメモリを分けたほうが効率よく計算できる．そこで，グラフ分割する際，物理計算ノード単位ではなく，CPU ソケットごとに分割したほうがよい．その場合，同じ物理ノードを共有するプロセス間では，共有メモリを使うことでデータを共有することが可能である．これを利用した，通信データ量削減を提案する．

$R \times C$ の２次元分割において，同じ列の計算ノードを，同じ物理計算ノードに配置すると，ボトムアップ探索の allgather 通信において，同じ物理ノードに配置されたプロセスは必ず同一通信グループとなる．そこで，各物理ノードに配置されたプロセスのうち１つだけがまとめて allgather 通信をすることで，通信データ量（あるいは，MPI 内部での無駄なメモリコピー）を削減することができる．TSUBAME2.5 は１物理ノードに２ソケット CPU があるので，最大で通信データ量を半分にすることができる．

4.2 ボトムアップ探索の２方向同時通信

「京」や BlueGene/Q など，３次元トーラスや４次元以上のトーラスメッシュネットワークを備えたスーパーコンピュータも多い．これらの直接網で接続されたネットワークでは，計算ノードに接続されたインターコネクトが複数あり，できるだけ多くの本数を使って通信したほうが，性能が出しやすい．そこで，ボトムアップの Rotate Completed 通信を，２方向同時に通信させることで通信の高速化を図った．図 11 にその方法を示す．

$c_{i,j}$ は通信するデータであり，s はステップ数である．このステップ数は 2C また 4C あり，偶数番目と奇数番目で通信方向を分けている．これにより，最大で２本の接続を各双方向使って通信することができる．

Function sendrecv-completed ($c_{i,j}$, s)
1. route ← s mod 2
2. **if** route = 0 **then**
3. | $c_{i,j}$ ← sendrecv($c_{i,j}$, $P(i,j+1)$, $P(i,j-1)$)
4. **else**
5. | $c_{i,j}$ ← sendrecv($c_{i,j}$, $P(i,j-1)$, $P(i,j+1)$)

図 11 ボトムアップ探索の２方向同時通信

5. 性能評価

本論文の提案手法を「京」および TSUBAME2.5 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．

5.1 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．

5.2 「京」と TSUBAME2.5

「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．

TSUBAME2.5 は東京工業大学に設置されているスーパーコンピュータで，各計算ノードに Intel CPU 6 コア 2 ソケットと 54GB のメモリがあり，ネットワークは Dual Rail の Infiniband Fat-tree である．各計算ノードは 4GB/s×2（双方向）の Infiniband 接続を２本備えており，合計 8GB/s×2（双方向）の通信が可能である．

5.3 最適化の効果

ボトムアップ探索の２方向同時通信の効果を評価した．図 12 は，「京」における最適化による通信待ち時間の変化

である．２方向同時通信により，１方向のみの場合に比べて通信待ち時間が減少していることが分かる．

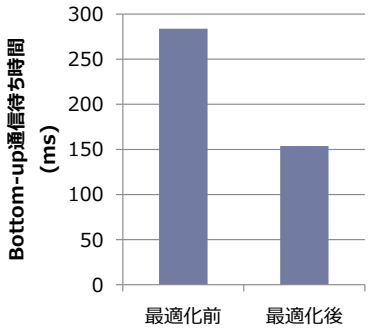

図 12 ボトムアップ探索の２方向同時通信（「京」65536 ノードで Scale 40 を計算した際のボトムアップ通信待ち時間）

5.4 ウィークスケーリング性能と実行時間内訳

次に，ウィークスケーリングによる性能評価を行った．図 13 がその結果である．64 ノードにおける性能を基準に，リニアにスケールした場合の性能と比較すると，1024 ノードで 56%，65536 ノードで 22%の性能であった．２次元分割による分割サイズが 64 ノードの場合 8x8, 1024 ノード場合 32x32，65536 ノードの場合 256x256 なので，ボトムアップ探索における通信データ量が，R,C に比例して大きくなることが大規模に実行した場合のウィークスケーリング性能を下げている要因である．しかし，提案手法は 65536 ノードでも性能が頭打ちにはなっていない．本論文の提案手法は，65536 ノードで 17,977GTEPS を達成した．

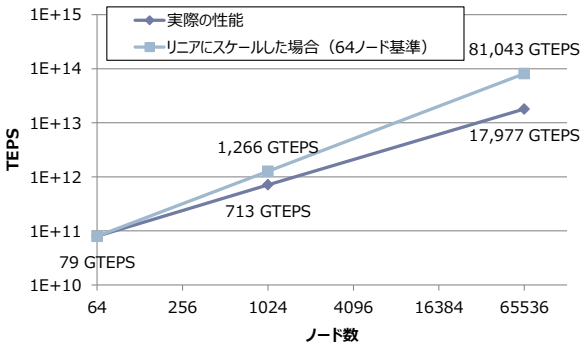

図 13 「京」におけるウィークスケーリング性能（問題サイズは 65536 ノードで Scale 40）

図 14 は実行時間内訳である．64 ノードと 65536 ノードを比較すると，通信時間の増加がそのまま全体の実行時間を上げているのが分かる．65536 ノードにおける通信待ち時間は，全体の実行時間の 73%であった．

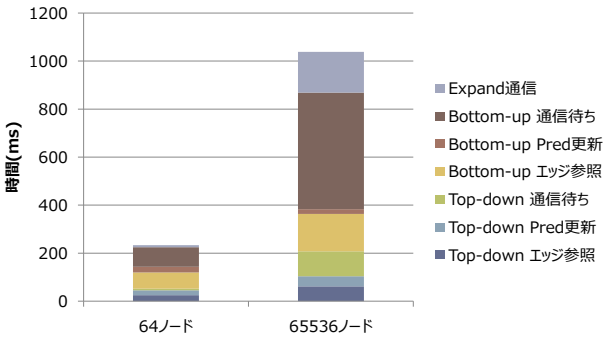

図 14 「京」64 ノード（Scale30），65536 ノード（Scale40）実行時の実行時間内訳

5.5 TSUBAME2.5 における性能

図 15 は TSUBAME2.5 における提案手法（2014 April）と，参考に示したトップダウンのみの手法（2012 September）のウィークスケーリング性能である．提案手法は，1024 ノードで 1,280 GTEPS であり，トップダウンのみの手法と比べて 3 倍程度の性能となっている．

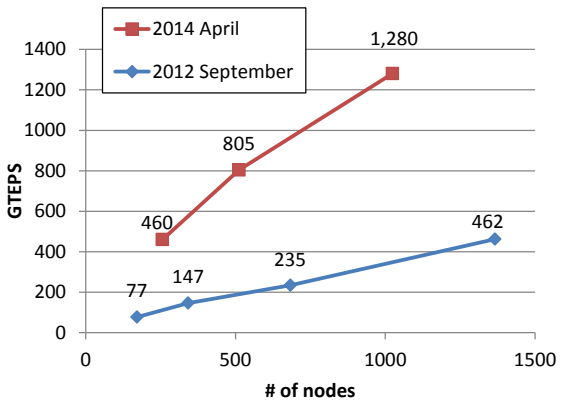

図 15 TSUBAME2.5 におけるハイブリッド BFS のウィークスケーリング性能（問題サイズは 1024 ノードで Scale 35）

6. 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，本研究における「京」65536 ノードでの性能に迫る 16,599GTEPS の性能を達成している．

7. 結論と今後の展望

比較的直径の短いグラフに対しては，トップダウンとボトムアップのハイブリッドで探索を行うハイブリッド BFS という手法が有効である．ハイブリッド BFS の並列分散アルゴリズムとして Beamer らによる隣接行列の２次元分割[3]を使った手法が提案されているが，数千～数万ノード規模に分割した場合，通信データ量や計算ステップ数が多くなる関係で性能が頭打ちになってしまっていた．そこで，ビットマップを使った疎行列表現や，頂点濃度に応じたデータ構造選択，ボトムアップ探索の並列性を上げることによる効率化を行い，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を提案した．「京」を使った性能評価では，65,536 ノードで 17,997GTEPS の性能を達成した．この性能により 2014 年 6 月の Graph500 ランキングにおいて「京」は１位を獲得している．

本論文で提案した各最適化のより詳細な性能評価，特に 3.2 節の頂点濃度に応じたデータ構造選択について性能評価は，今後の課題とする．また，現在ボトルネックになっている通信のさらなる最適化を今後行う予定である．

謝辞　本研究は JST CREST の支援により行われたものである．また，本研究の性能評価では，TSUBAME グランドチャレンジ大規模計算制度を利用した．

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.