Vol.2011-ARC-197 No.20
Vol.2011-HPC-132 No.20
2011/11/29

大規模グラフ処理ベンチマーク Graph500 への２次元分割の適用と性能評価

上野 晃司† 鈴村 豊太郎††

Graph 500 とは，スーパーコンピュータのグラフ処理性能を測定する新しいベンチマークである．スパコンのベンチマークでは，数値計算性能を測る Linpack による Top 500 が有名だが，近年，大規模グラフ処理が，重要性を増しており，Graph500 ベンチマークが広がりを見せている．Graph500 のリファレンス実装は，使用されているアルゴリズムの問題により，分散メモリ環境で大規模にスケールさせることができなかった．そこで，大規模にスケール可能な２次元分割に注目した．本論文では，２次元分割を TSUBAME2.0 上に実装し，512 ノードで頂点数 2^{36}（68.7 billion），枝数 2^{40}（1.1 trillion）のグラフ（Graph500 の Scale 36）の BFS(幅優先探索)を 24.12 秒で計算した．TEPS 値は 22.8GE/s であり，２次元分割による実装が大規模に分散可能であることが分かった．また，クラスタ型スパコンでは一般的な InfiniBand ネットワーク環境は，大規模グラフ処理に適していることが分かった．

Performance Evaluation of the Large-Scale Graph Analysis Benchmark, Graph500 with 2D Partitioning Method

Koji Ueno† and Toyotaro Suzumura††

Graph500 is a new benchmark that ranks supercomputers by executing a large-scale graph search problem. Our early study reveals that the provided reference implementations are not scalable in a large-scale distributed environment. In this paper we implement an optimized method based on 2D based partitioning. Our implementation can solve BFS (Breadth First Search) of large-scale graph with 2^{36} (68.7 billion) vertices and 240 (1.1 trillion) edges for 24.12 seconds with 512 nodes and 12288 CPU cores. This record corresponds to 22.8 GE/s. We found 2D based partitioning method is scalable for large-scale distributed systems. We also demonstrate thorough study of performance characteristics of our optimized implementation and reference implementations in a large-scale distributed memory supercomputer with the Fat-Tree based Infiniband network.

1. はじめに

大規模グラフ処理は Web ページのリンク解析，タンパク質間の相互作用解析，VLSI のレイアウトや道路網，送電網の最適化など様々な応用分野あり，近年盛んに研究されている．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，大規模グラフ処理も重要なアプリケーションとなりつつあり，そのような中，スパコンのグラフ処理性能を測定する，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．Graph 500 は，スパコンの通信性能や，グラフデータを格納するメモリの大きさや，メモリへのランダムアクセス性能を測るという，データインテンシブなベンチマークであり，数値計算性能を測る Top 500 ベンチマークとは計測する性能が全く異なる．

Graph500 は新しいベンチマークであり，最適化に関しての研究はまだ少ない．我々は[15]において，リファレンス実装のアルゴリズムや性能の解析を行っている．そこで得られた知見に基づいて，本論文では２次元分割による隣接行列の分割方法[2]に着目し，大規模分散環境でスケール可能な Graph500 の実装方法を提案する．また，分散メモリ環境で大規模に実行した場合の性能特性を調査した．本論文の貢献は，

1. ２次元分割による分散 BFS を TSUBAME2.0 上に実装した．
2. TSUBAME2.0 で大規模に実行し，2 次元分割による実装とリファレンス実装の性能を比較した．
3. ２次元分割による分散 BFS は，大規模に分散可能であることを実証した．

以降，2 章は Graph500 ベンチマークの概要と，分散 BFS アルゴリズムについて，3 章はリファレンス実装の大規模分散環境におけるスケーラビリティの問題について，4 章は 2 次元分割による分散 BFS アルゴリズムとその実装について，5 章は 2 次元分割による実装とリファレンス実装の性能を比較，6 章はその考察，7 章は関連研究，8 章はまとめと今後の展望である．

2. Graph500 ベンチマークとBFSアルゴリズム

この章では，Graph500 ベンチマークの概要と，分散 BFS アルゴリズムについて述べる．さらに，分散 BFS アルゴリズムの行列ベクトル積との関係についても説明する．

2.1 Graph500 ベンチマークの概要

Graph500 は，大規模なグラフに対して BFS による探索を実行するベンチマークで

† 東京工業大学
Tokyo Institute of Technology
†† 東京工業大学／IBM 東京基礎研究所
Tokyo Institute of Technology / IBM-Research Tokyo

ⓒ 2011 Information Processing Society of Japan

ある．単位時間に処理できた枝数と，扱える最大問題サイズが評価指標となる．計算インテンシブな Top500 ベンチマークと違い，Graph500 ベンチマークは，データインテンシブなベンチマークである．扱える問題サイズは，グラフの頂点数=2^{SCALE} であるような SCALE 値で表す．単位時間に処理できた枝数は，TEPS (Traversed Edges Per Second) 値で表す．例えば，100 万 TEPS とは，100 万個の枝を持つ連結グラフの BFS が 1 秒で完了した場合の性能である．

ベンチマークを実行するプログラムは，(a)グラフデータの生成，(b)計算するのに最適なデータ構造への変換，(c)BFS による探索，(d)計算結果の検証の 4 つの部分から成る．ベンチマークの実行順は次のようになっている．最初に(a),(b)によりグラフデータを構築し，グラフから始点を 64 個選ぶ．次に，64 個の始点それぞれに対して順番に，(c)BFS による探索と，(d)計算結果の検証を行う．複数の始点からの探索を同時に行うことはできない．時間を計測しベンチマークとする部分は，(b)のグラフデータ構造への変換（Kernel 1）と，(c)の BFS による探索（Kernel 2）のみである．(a)では，枝数が頂点数の 16 倍となるようなクロネッカーグラフ[3]を生成する．枝はすべて重みなし，無向辺である．ここで生成されるデータは規則性のない順番で並んだ，枝のリストである．(b)では(a)で生成された枝リストから，隣接行列の CSR (Compressed Sparse Row)や，CSC (Compressed Sparse Column)などのグラフデータ構造に変換する．(c)の BFS は，BFS で辿った頂点の軌跡である BFS 木を出力する．(d)では，この BFS 木が正しいかどうかチェックする．このチェックでは，BFS 木にループがないこと，枝の張っている頂点同士の深さの差が 1 以下であること，などの 5 つのルールを満たしていることをチェックする．

Graph500 リストの発表は，2011 年 11 月のリストで 3 回目となるが，3 回目からリストの集計方法が変わり，2 回目までは問題サイズでランキングを決めていたが，3 回目からは，TEPS 値でランキングを決めるようになった．

2.2 Level-synchronized BFS

Graph500 のリファレンス実装には，OpenMP で書かれた実装や，MPI で書かれた実装，Cray の共有メモリ型プログラミング環境用の実装，など複数の種類が用意されている．TSUBAME2.0 で分散実行するには，MPI で書かれた実装を使用する．MPI で書かれた実装には，さらにアルゴリズムや実装方法の異なる 4 種類の実装が用意されている．これらの実装は，対象としているプログラミング環境や分散方法などは異なるが，全てベースとなるアルゴリズムとして Level-synchronized BFS を使っている．このアルゴリズムは，各レベル（深さ）について，そのレベルの頂点をすべて処理してから，次のレベルに進むというアルゴリズムである．

アルゴリズム 1 : Level-synchronized BFS

```
1    for all vertex v in parallel do
2    | PRED[v]← -1;
3    |   VISITED [v]  ←  0;

4    PRED [r] ← 0
5    VISITED[v]  ←  1
6    Enqueue(CQ , r)
7    While CQ != Empty do
8    |   NQ ← empty
9    |   for all u in CQ in parallel do
10   |   |  u ← Dequeue(CQ)
11   |   |  for each v adjacent to u in parallel do
12   |   |   | if VISITED [v] = 0 then
13   |   |   |   |   VISITED [v] ← 1;
14   |   |   |   |   PRED [v] ← u;
15   |   |   |   |   Enqueue(NQ, v)
16   |   swap(CQ, NQ);
```

アルゴリズム 1 は Level-Synchronized BFS の擬似コードである．まず，BFS 木を格納する PRED と，頂点が訪問済みかどうかを格納する VISITED を初期化する．PRED[v]は頂点 v の BFS 木における親頂点を表す．初期値-1 は BFS 木にまだ入っていないことを表している．VISITED[v]は頂点 v が訪問済みかどうかを表す．初期値 0 はまだ訪問していないことを表している．次に，BFS の始点となる頂点を CQ (Current Queue) に入れ，探索を開始する．

探索においては，7~16 行の 1 ループが 1 レベルに相当する．このループ中で，CQ は現在のレベルで訪問する頂点，NQ (Next Queue) は次のレベルで訪問する頂点が格納されている．例えば，レベル 1 で CQ に頂点γが入っていたとすると，11，12 行目でγの隣接頂点が訪問済みかどうかチェックされ，まだ訪問していない頂点は NQ に格納される．レベル 2 では、CQ にはこれらの頂点が格納されていることになる．9 行目の for と，11 行目の for は並列化が可能なループである．

リファレンス実装の 4 つの MPI 実装は，基本的には Level-synchronized BFS を実装しているが，グラフデータの分散方法などに違いがある．4 つのリファレンス MPI 実装の処理の仕方の違いを簡単に理解するため，次に Level-synchronized BFS と疎行列ベクトル積の関係について説明する．

2.3 Level-synchronized BFSと疎行列ベクトル積

Level-synchronized BFS の分散処理は，行列ベクトル積の分散処理と似ている．行列ベクトル積とは，x, y をベクトル，A を行列として，y=Ax を計算することである．こ

© 2011 Information Processing Society of Japan

こで，A をグラフの隣接行列とする．要素の値は，対応する枝がある場合 1，ない場合 0 である．また，x を CQ (Current Queue)に相当するベクトルとする．頂点 v∈CQ なら x(v)=1，そうでないなら x(v)=0 である．ここで，x(v) はベクトル x の v 番目の値である．すると，行列ベクトル積の結果であるベクトル y から，CQ に入っている各頂点の、隣接頂点すべて（以下、CQ の隣接頂点）が分かる．これは，頂点 v に対応する値 y(v) がゼロでなければ，頂点 v は CQ の隣接頂点となっているからである．CQ の隣接頂点が分かるということは，アルゴリズム 1 の 11 行目の頂点 v が分かるということなので，この行列ベクトル積は，アルゴリズム 1 の 9~11 行目と同じ計算をしたことになる．実際，リファレンス MPI 実装は，行列ベクトル積と同じような方法で，CQ の隣接頂点を計算している．

行列ベクトル積は容易に並列化が可能な問題であり，様々な並列化方法が考えられる．リファレンス MPI 実装には，replicated-csr， replicated-csc，simple，one_sided の 4 つが用意されている． 4 つのリファレンス実装が使用しているアルゴリズムは，隣接行列の分割方法の違いで，大きく 2 つに分けることができる． 1 つは隣接行列を縦に分割する方法(replicated-csr, replicated-csc が該当，図 1 の左)，もう 1 つは横に分割する方法(simple, one_sided が該当，図 1 の右)であり．図は，P 個のプロセッサがある場合の、2 つの分割方法による Level synchronized BFS を行列ベクトル積として表したものである．プロセッサ k は部分隣接行列 A_k を持っている．隣接行列の他の部分は，他のプロセッサが持っているので，プロセッサ k は持っていない．図中のベクトル x は CQ に相当するものである．

$$\begin{bmatrix} A_1 \\ \hline A_2 \\ \hline \vdots \\ \hline A_P \end{bmatrix} \times x \qquad \left[\begin{array}{c|c|c|c} A_1 & A_2 & \cdots & A_P \end{array}\right] \times x$$

図 1 行列を縦 P 個に分割（左），行列を横 P 個に分割（右）

3. リファレンス実装のスケーラビリティにおける問題

この章では， 4 つのリファレンス MPI 実装（replicated-csr， replicated-csc，simple，one_sided）のアルゴリズムを説明し，そのアルゴリズムによって引き起こされるスケーラビリティの問題について述べる．

3.1 リファレンス実装: Replicated-csr, Replicated-csc

隣接行列を縦に分割する replicated-csr や replicated-csc（以下，両方合わせて replicated とする）では，CQ 全体を，すべてのプロセッサにコピーする．各プロセッサは，自分が持っている部分隣接行列を使って，CQ の隣接頂点を探し，自分の担当する領域の PRED, NQ を計算する．

CQ を，すべてのプロセッサにコピーするというのは，つまり，分散して持っている NQ を，他のすべてのプロセッサに送信するということである．CQ や NQ は 1 頂点につき 1 ビットで表すことが可能なため，データ量は大きくなく，分散数が小さい場合には，通信データ量を低く抑えることができ，有効な計算方法となる．ただし，CQ の大きさは，グラフ全体の頂点数に比例するので， 1 プロセスあたりの通信データ量はグラフ全体の頂点数に比例する．よって，分散数が非常に大きな場合には，このコピーで膨大な通信が必要になり，問題となる．

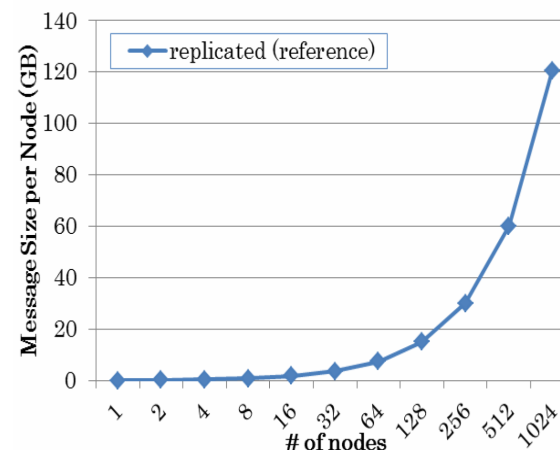

図 2 weak-scaling における replicated の 1 ノードあたりの通信データ量

図 2 は replicated の 1 ノードあたりの通信データ量である．問題サイズは weak-scaling で 1 ノードあたり Scale 26． 1 ノードあたり 2 プロセスで計算した場合の理論値である．Weak-scaling では，ノード数の増加に比例して，問題サイズが増加するので，頂点数も増加する． 1 ノードあたりの通信データ量はグラフ全体の頂点数に比例して増えるので，図 2 のようになる．よって，スケールしないことは容易に想像できる．

© 2011 Information Processing Society of Japan

3.2 リファレンス実装: simple, one_sided

隣接行列を横に分割する simple では，CQ を P 個に分割して配置しておく．NQ はもともと P 分割されているので，前のレベルで計算された NQ をそのまま CQ として使えば良い．各プロセッサは，分割された CQ と，自分の持っている部分隣接行列から，CQ の隣接頂点を探す．CQ の隣接頂点を使って PRED や NQ を更新するが，ここで見つかった CQ の隣接頂点には，自分が担当する頂点もあれば，他のプロセッサが担当する頂点もある．自分が担当する頂点は自分で処理し，他のプロセッサが担当する頂点は，そのプロセッサに送信し，処理してもらう．ここで送信される CQ の隣接頂点の数は，最大で，送信元のプロセッサが持っている部分隣接行列の枝数までとなる．よって，通信データ量はノード数が十分大きい場合は，ノード数に関係なく一定となる．ただし，縦に分割する replicated では通信するデータ(CQ)は 1 頂点につき 1 ビットであったのに対し，横に分割する simple では，CQ の頂点と，隣接頂点との組みを送信しなければならない．これは，PRED の更新には親頂点（CQ の頂点）が必要となるためである．そのため，分散数が少ない場合だと，縦に分割する replicated の方が通信データ量は少なくなるので，replicated の方が有利である．

one_sided は通信に MPI の one_sided 操作を使用する実装であるが，アルゴリズムは simple と同じである．

隣接行列を横に分割する，simple, one_sided は，replicated のような通信データ量の問題はない．しかし，CQ の隣接頂点を担当頂点の送信するところで，すべてのノードが他のすべてのノードに異なるデータを送信する全対全通信が必要になる．この通信は大規模に分散させた場合，スケールさせることが難しい．これは、次のようなマイクロベンチマークの結果から明らかである．

図 3 は，TSUBAME2.0 で全対全通信を行った場合の，通信速度である．1 ノードあたり 4MPI プロセス．MPI 実装は MVAPICH2 1.6[4]である．通信は MPI_Alltoallv を使い，この関数の送信バッファに 64, 256, 1024MB と 3 種類の大きさのバッファを入力させた．各ノードが他の各ノードに送信するデータ量は，バッファが例えば 1024MB で，MPI プロセス数が 256（この場合，ノード数は 64）の場合，4MB になる．図から，全対全通信がスケールしていないことが分かる．512 ノードでは，64MB などの小さいバッファでは極端に遅くなり，また，1024MB という大きなバッファを用意しても，32 ノードの場合に比べて速度は 1/4 以下に低下している．TSUBAME2.0 のネットワークは Infiniband による Fat-Tree で，理論ピーク性能が達成されれば，すべてのノードが全対全通信を行った場合でも，速度低下は発生しない．よって，ソフトウェアオーバヘッドによる速度低下が大きく影響していると考えられる．

したがって、全対全通信が必要になる simple, one_sided はスケールしない。

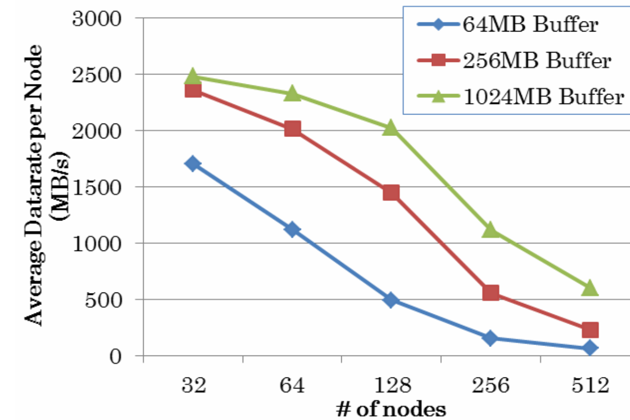

図 3　TSUBAME2.0 で全対全通信を行った時の通信速度

4. 2 次元分割によるスケーラブルな実装

リファレンス実装の 2 つの分割方法はどちらも、隣接行列を 1 次元で分割した方法である。前章で述べたとおり、1 次元分割の 2 つの分割方法はどちらもスケールさせることが難しい．そこで，隣接行列を 2 次元に分割するアルゴリズム（2 次元分割）[2] を実装した．

4.1 ２次元分割による分散BFSアルゴリズム

プロセッサを $P = R \times C$ の 2 次元メッシュ(mesh)に配置する．このメッシュの行を「プロセッサ行」，列を「プロセッサ列」と呼ぶことにする．隣接行列を図 4 のように $R \times C$ 個の行と C 個の列に分割し，プロセッサ (i, j) は，隣接行列の $A^{(1)}_{i,j} \sim A^{(C)}_{i,j}$ の C ブロックを担当する．頂点は，$R \times C$ 個のブロックに分割し，プロセッサ (i, j) は，$j \times R + i$ 番目のブロックを担当する．

1 レベルにつき，Expand と Fold と呼ばれる 2 段階の通信を行う．各レベルで行う操作について説明する。まず、各プロセッサは自分の担当する頂点ブロックの CQ を同じプロセッサ列の他のプロセッサに送信する．これを Expand という．Expand は 1 次元分割の縦分割と同じように，CQ をコピーする通信であるが，隣接行列は横にも C 個に分割されているので，通信は，同じプロセッサ列の他のプロセッサとだけ行う．次に，各プロセッサは CQ と各プロセッサが持っている部分隣接行列から，CQ の隣

ⓒ 2011 Information Processing Society of Japan

接頂点を探し、PRED や NQ を更新するため，CQ の隣接頂点を，その頂点の担当プロセッサに送信する．この通信を Fold という．PRED を更新するのに，親の頂点が必要なので，Fold では，CQ の隣接頂点と，親頂点（CQ の頂点）の組みを送信することになる．Fold は１次元分割の横分割と同じように，CQ の隣接頂点を担当プロセッサに送信する通信だ．しかし，２次元分割では，隣接行列の分割方法から，Fold の通信を行う必要のある相手は, 同じプロセッサ行の他のプロセッサのみとなる. このように，１次元分割は２種類の分割方法を合わせた方法であり，C=1 で縦分割（図１の左），R=1 で横分割（図１の右）と同じになる．

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

図４　隣接行列の２次元分割

２次元分割の利点は，通信で絡むプロセッサ数が少ないことである．１次元分割では，２種類の分割方法のどちらも，全対全の通信が必要だったのに対し，２次元分割の場合, Expand では同じ列のノード(R-1)プロセッサと, Fold では同じ行のノード(C-1)プロセッサとしか通信を行わない．よって，通信するプロセッサ数を少なくすることができ，大規模に分散可能になる．

4.2 実装方法

Expand の通信は，CQ を MPI の Allgather を使って実装した．これは，リファレンス実装の replicated でも使われている方法である．Fold の通信では，CQ の隣接頂点を探し，送信する送信側と，頂点を受信し VISITED や PRED, NQ を更新する受信側との２つのタスクに分解し，これらを並列化した．さらに，通信を非同期で行っている．これらの工夫により高効率な処理を実現した．アルゴリズム２は，実装したアルゴリズムの擬似コードである．

アルゴリズム２: 2次元分割によるBFSの実装

```
1   for all vertex lu in NQ do
2   |  NQ[lu] ← 0
3   NQ [root] ← 1
4   fork;
5   for level = 1 to ∞
6   |  CQ ← all gather NQ in this processor-column;
7   |  parallel Task A and Task B
8   |  Synchronize;
9   |  if NQ = φ for all processors then
10  |  |  terminate loop;
11  join;

Task A (送信側)
1   for all vertex gu in CQ parallel do (contiguous access)
2   |  if CQ [gu] = 1 then
3   |     for each local vertex v adjacent to gu do
4   |     |  send tuple (gu, v) to the processor which owns vertex v

Task B (受信側)
1   for all received tuple (gu, v) parallel do
2   |  if visited[v] = 0 then
3   |  |  PRED[v] ← gu;
4   |  |  VISITED[v] ← 1;
5   |  |  NQ [v] ← 1;
```

また，OpenMP を使ったプロセス内のマルチスレッド化も行なっていて，MPI と OpenMP のハイブリッド並列で実装した．

5. 性能評価

この章では, 東工大のスパコン TSUBAME2.0 上での性能評価の結果について述べる. TSUBAME2.0 は, 1400 以上のノードが Fat-Tree によるフルバイセクションの Infiniband ネットワークで接続されている．各ノードには，Intel CPU Xeon 5670 2.93GHz

ⓒ 2011 Information Processing Society of Japan

（Westmere EP,，6 コア，256-KB L2 キャッシュ, 12-MB L3) が２つ，NVIDIA M2050 GPU (Fermi) が３つ, 48GB のメモリが搭載されている．通信は，各ノードは Infiniband QDR が２リンク使用可能で，合計 80Gbps の通信バンド幅を備えている．

最大 512 ノードまで使用して実験した．なお, GPU は使用していない．TSUBAME2.0 は 1 ノードあたり物理コア 12 個だが，SMT を有効にすると仮想的に 24 コアになる．1 ノード 24 コアとして，各プロセスに均等に割り振った．gcc 4.3.4 (OpenMP 2.5), MVAPICH2 1.6 [4]を使用した．比較するリファレンス実装は，執筆時点で最新の version 2.1.4 である．

以下の実験では，２次元分割のプロセッサ配置 R×C は表のようにした．R と C の値は、通信データ量の関係からなるべく近い値になるようにしてある．また、R,C は使用ノード数とは関係無く，MPI プロセス数から決定した．

プロセス数	1	2	4	8	16	32	64	128	256	512	1024
R	1	1	2	2	4	4	8	8	16	16	32
C	1	2	2	4	4	8	8	16	16	32	32

また，問題サイズは，すべて weak-scaling で 1 ノードあたり Scale 26 とした．つまり，実行ノード数が，例えば 1 ノードの場合 Scale 26，２ノードの場合 Scale 27，４ノードの場合 Scale 28 である．この問題サイズは、ディスクなどのストレージを使用しない場合の最大サイズである．

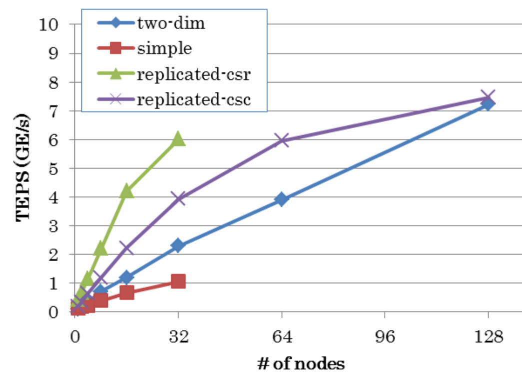

図 5　２次元分割とリファレンス実装を比較（1～128 ノード）

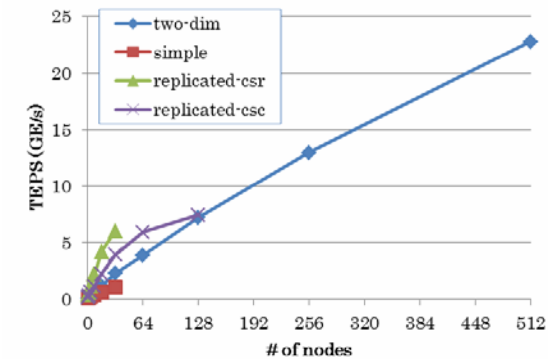

図 6　２次元分割とリファレンス実装を比較（1～512 ノード）

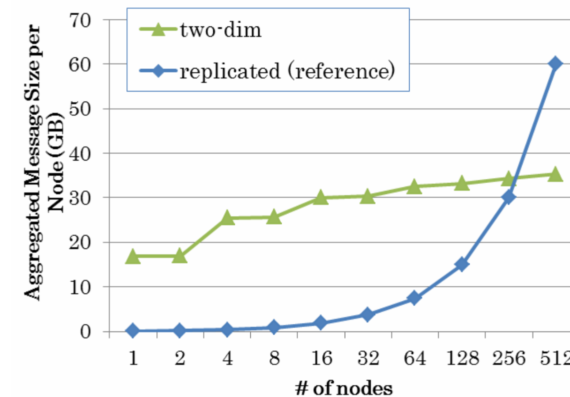

図 7　１ノードあたりの通信データ量の比較

図 5，6 は２次元分割とリファレンス実装の比較である．横軸はノード数で縦軸は TEPS (GE/s)である．リファレンス実装の replicated-csr, replicated-csc と２次元分割（two-dim）は，１ノードあたり 2MPI プロセスで実行し，リファレンス実装の simple は１ノードあたり 16MPI プロセスで実行した．リファレンス実装の one_sided は、細粒度の one sided 操作を頻繁に実行する実装になっているが、使用した MPI 実装が、このような操作に最適化されていないため、極端に性能が低くなる。そのため、リファレンス実装の one_sided は実験から除外した。図 7 は１ノードあたりの通信データ

ⓒ 2011 Information Processing Society of Japan

量を、replicated と２次元分割で比較したグラフである．replicated の通信データ量は理論値を算出した．

リファレンス実装で，データがない部分はエラーなどで計測できなったところである．Simple はノード数を大きくするとメモリ不足エラーとなり，Replicated-csr は Scale 32 で validation エラーとなり，Scale 33 以上で segmentation fault，Replicated-csc は Scale 34 以上で construction 時にエラーとなる．原因の詳細については掴めていない．

２次元分割の実装は，リファレンス実装の simple の２倍程度の速度が出ている．これは，送信処理と受信処理の並列化や，OpenMP によるプロセス内の並列化の効果によるものである．２次元分割の実装は，リファレンス実装の replicated と比べると，性能が低い．ただし，これは 3.1 章で述べた通り，replicated のアルゴリズムはノード数が小さい場合には通信データ量を小さくすることができ，有利だからである．図８から分かるように，replicated の優位性もノード数が増えるにしたがって急激に低下し，通信データ量は 512 ノードで２次元分割と逆転する．実際，図５から replicated-csc はノード数 128 で既に性能の限界が見え始めている．図７から，２次元分割は，ノード数の増加に従って徐々に１ノードあたりの通信データ量は大きくなるが，その増加幅は小さく，十分スケール可能であることが分かる．

２次元分割は，512 ノードで，Scale 36 を計算し，TEPS 値 22.8GE/s の性能を達成した． 512 ノードでこれだけの性能を出す TSUBAME2.0 は，Graph500 リストの他のスパコンと比較してグラフ処理に向いていると言える．

6. 議論

TSUBAME2.0 は，他のスパコンと比較してグラフ処理ベンチマークで高いスコアを出しやすい理由は，ネットワークトポロジにある．物理シミュレーションは，隣接ノード間の通信が重視されることが多いが，グラフ処理は離れたノード間の通信速度も重要である．Graph500 リストの上位にあるマシンの多くは，BlueGene や Cray など，ネットワークトポロジが３次元トーラスのマシンが多いが，３次元トーラスのマシンは離れたノード間の通信はコストが大きい．TSUBAME2.0 のネットワークトポロジは Fat-tree であり、離れたノード間の通信が隣接ノード間とほぼ変わらないので、グラフ処理に向いている．

図８は，全ノード合計の通信データ量を BFS の実行時間で割った，平均通信データレートである．実際に、512 ノードまで，ほぼリニアに通信速度が上がっているのが分かる．512 ノード使用した場合でも，１ノードあたり 1.4GB/s を超える速度が出ている．TSUBAME2.0 がグラフ処理に向いているということが分かる．

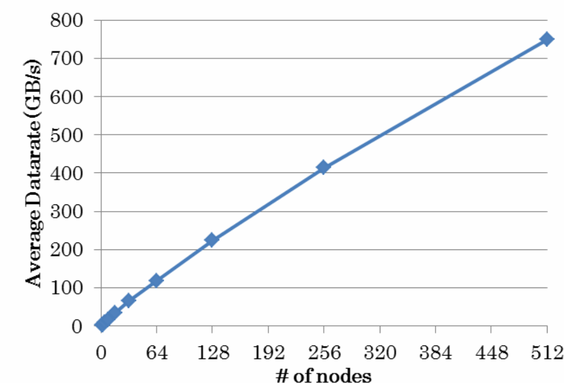

図８　全ノード合計の平均通信データレート

7. 関連研究

グラフ処理の中でも，基本となる BFS は，様々なマシンへの最適化が研究されている．本研究で使用した２次元分割は，Andy Yoo ら[2]が元々は BlueGene/L に実装したアルゴリズムであった．しかし，彼らの提案した実装は，通信データ量の削減が重要視されていて，BFS が十分に最適化されているとは言えない．David A.ら[5]は，共有メモリ型のマシンである Cray MTA-2 に BFS と st-connectivity を最適化した．Virat Agarwal ら[6]は，BFS を Intel Nehalem CPU による共有メモリ型マルチプロセッサに最適化した．Guojing Cong ら[7]は，BFS を PGAS 言語上で最適化した．

また、グラフ処理の高速化に関する研究には、アクセラレータである GPU への最適化 [12,13] や Cell/BE への最適化 [14] も研究されている。

大規模グラフ処理を一般化し，様々なアルゴリズムを処理することができる処理系として、Pregel [8] や、疎行列ベクトル積で表せるグラフ処理を Hadoop 上で実現した PEGASUS [9]，GBASE[10] なども提案されている。

8. まとめと今後の展望

本論文では Graph500 ベンチマークを大規模分散環境でスケールさせるため，２次元分割による BFS を実装した．Graph500 のリファレンスコードは，大規模分散環境ではスケールしない．２次元分割による実装は，TSUBAME2.0 の 512 ノードで Scale 36 を計算し，TEPS 値 22.8GE/s を達成した．これにより，2 次元分割なら大規模にスケールさせることが可能だということが分かった．

ⓒ 2011 Information Processing Society of Japan

我々は執筆時点ですでに，２次元分割の実装に通信データの圧縮や頂点の並び替えなどの最適化により，TSUBAME2.0 を 1366 ノード使用した実験で，TEPS 値 103GE/s を達成している．この性能は執筆時点で最新の Graph500 リスト（June 2011）TEPS 値ランキング１位のスコアの２倍を超える性能である．これらの成果の詳細は，また別の機会に発表する．

謝辞　本研究は，科学技術振興機構(JST) の戦略的創造研究推進事業「CREST」における研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」によるものである。また、TSUBAME2.0 グランドチャレンジに協力していただいた方々に感謝の意を表する．

参考文献

1) Graph500 : http://www.graph500.org/
2) 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.
3) 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.
4) MVAPICH2: http://mvapich.cse.ohio-state.edu/
5) David A. Bader and Kamesh Madduri. 2006. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06). IEEE Computer Society, Washington, DC, USA, 523-530
6) Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable Graph Exploration on Multicore Processors. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11.
7) Guojing Cong, George Almasi, and Vijay Saraswat. 2010. Fast PGAS Implementation of Distributed Graph Algorithms. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11.
8) Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 international conference on Management of data (SIGMOD '10). ACM, New York, NY, USA, 135-146.
9) U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM '09). IEEE Computer Society, Washington, DC, USA, 229-238.
10) U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: a scalable and general graph management system. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '11). ACM, New York, NY, USA, 1091-1099.
11) Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proceedings of the 14th international conference on High performance computing (HiPC'07), Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna (Eds.). Springer-Verlag, Berlin, Heidelberg, 197-208.
12) Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In Proceedings of the 14th international conference on High performance computing (HiPC'07), Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna (Eds.). Springer-Verlag, Berlin, Heidelberg, 197-208.
13) Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2011. PHAST: Hardware-Accelerated Shortest Path Trees. In Proceedings of Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. Anchorage, AK, USA, 921 – 931.
14) Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. 2008. Efficient Breadth-First Search on the Cell/BE Processor. IEEE Trans. Parallel Distrib. Syst. 19, 10 (October 2008), 1381-1395.
15) Toyotaro Suzumura, Koji Ueno, Hitoshi Sato, Katsuki Fujisawa and Satoshi Matsuoka, "Performance Evaluation of Graph500 on Large-Scale Distributed Environment", IEEE IISWC 2011 (IEEE International Symposium on Workload Characterization) , 2011/11, Austin, TX, US

© 2011 Information Processing Society of Japan

正誤表

訂正箇所	誤	正
p1 の論文著者	鈴村　豊大郎	鈴村　豊太郎

ⓒ2011 Information Processing Society of Japan