Vol.2013-HPC-139 No.12
2013/5/30

GPU クラスタにおける幅優先探索の高速化

平櫛 貴章[1,a)]　高橋 大介[2,b)]

概要：近年，様々な分野で巨大なグラフが出現しており，そのようなグラフを高速に処理する方法が必要となりつつある．また，GPU を搭載したクラスタシステムの台頭も著しく，Top500 ランキングにおいても複数の GPU クラスタが上位にランクインしている．しかし，LINPACK ベンチマークで示された性能に対して GPU クラスタのグラフ処理能力はあまり高いものとなっておらず，アルゴリズムの改善によるさらなる高速化が必要であると考えられる．そこで，本稿では GPU クラスタにおいて大規模なグラフの幅優先探索を高速化する手法を提案し，実装および評価を行った．その結果，GPU を利用することで CPU のみを用いた場合に比べてより高速に幅優先探索を行うことができることが分かった．

1. はじめに

グラフは広く用いられているデータ構造の一つである．グラフに対する処理は応用の幅も広く，交通網やソーシャルグラフなど，現実世界にあるものをグラフ化して処理することで有益な情報を得ることができる．これらの応用の中には非常に大規模なグラフを扱う必要のある課題もあるが，既存のベンチマークではうまく計算機システムのグラフ処理性能を評価できていなかった．このような背景から，Graph500 [1] と呼ばれるグラフ処理能力を競うベンチマークが 2010 年より実施されている．しかし，大規模グラフ処理においてはまだどのようなシステムやアルゴリズムが適しているのかはっきりしていない部分も多く，これらの差も大きな性能差の要因となっている．

また，近年ではアクセラレータとして GPU を多数搭載する計算機システムが増加しつつある．アクセラレータを搭載したシステムにおいては，それをうまく活用することがシステムの性能を引き出す上で重要となる．

Hybrid-BFS [2] は，現実世界に存在するネットワークにおいてよく見られる性質であるスモールワールド性を持つグラフ上で効率よく幅優先探索を行うことができるアルゴリズムである．このアルゴリズムのクラスタシステム向けの実装はすでに提案されているが [3]，GPU を用いた実装はまだ提案されていない．そこで，本稿では Hybrid-BFS における探索処理部分に GPU を活用することによる高速化手法を提案し，その評価を行った．その結果，クラスタシステム向けの Hybrid-BFS において GPU を利用することでより高速に探索を行うことができることが分かった．

[1] 筑波大学大学院システム情報工学研究科
[2] 筑波大学システム情報系
[a)] hiragushi@hpcs.cs.tsukuba.ac.jp
[b)] daisuke@cs.tsukuba.ac.jp

2. Hybrid-BFS

並列計算機における幅優先探索のアプローチとして，Level-synchronized BFS と呼ばれる手法がある．この手法では，ある頂点集合から辺を 1 つたどることで新たに訪問可能となる頂点の集合を求めて次の処理の入力とするという処理を繰り返すことで幅優先探索を行う．なお，処理を並列化する際には頂点集合から新たな頂点集合を求める部分を並列化する．

Hybrid-BFS もこの手法を元にしており，Top-down アプローチと Bottom-up アプローチと呼ばれる 2 つの手法をを適宜切り替えつつ組み合わせることによって，より高速な探索処理を実現している．ここでは，Hybrid-BFS の動作について述べる．

2.1 Top-down アプローチ

Top-down アプローチは以前より頻繁に用いられていた幅優先探索のためのアプローチで，すでに訪問済みの頂点と隣接している頂点のうちまだ訪問していない頂点を列挙する処理を繰り返すことで探索を行う．Top-down アプローチによる探索アルゴリズムを**図 1** に示す．

このアプローチでは，探索結果にあたるラベルデータと別に，訪問状況のみをビットマップとして保持することでキャッシュヒット率を向上させて探索性能を向上させることができる [4]．ただし，並列に探索処理を行う場合にはアトミック演算を用いたり，データの不整合を別のデータから補うなどの手法 [5] が必要となる．

© 2013 Information Processing Society of Japan

```
function top-down-step (frontier, predecessors)
  next ← ∅
  for u ∈ frontier do
    for v ∈ neighbors(u) do
      if predecessors[v] = -1 then
        predecessors[v] ← u
        next ← next ∪ { u }
      end if
    end for
  end for
  return next
```

図 1 Top-down アプローチによる探索アルゴリズム [2]

```
function bottom-up-step (frontier, predecessors)
  next ← ∅
  for v ∈ vertices do
    if predecessors[v] = -1 then
      for u ∈ neighbors(v) do
        if predecessors[u] ≠ -1 then
          predecessors[v] ← u
          next ← next ∪ u
          break
        end if
      end for
    end if
  end for
  return next
```

図 2 Bottom-up アプローチによる探索アルゴリズム [2]

2.2 Bottom-up アプローチ

Bottom-up アプローチでは，まだ訪問していない頂点からすでに訪問済みとなっている頂点と隣接しているものを列挙する処理を繰り返すことで探索を行う．Bottom-up アプローチによる探索アルゴリズムを**図 2** に示す．

このアプローチでは，各頂点が訪問可能になるかを判定するための処理を訪問可能と分かった時点ですぐに打ち切ることができるため，新たに訪問可能となる頂点が多い場合には Top-down アプローチと比べて高速に処理を行うことができる．また，Top-down アプローチと同様に訪問状態の管理にビットマップを用いる場合，Bottom-up アプローチではある程度の数の連続したインデックスを持つ頂点を同じスレッドが処理するようにすることで，アトミック演算やラベルデータを用いずに訪問状態を正しく管理することができる．

2.3 アプローチの切り替え

Hybrid-BFS では，2 つのアプローチのうちどちらを使うかをヒューリスティックを用いて決定している．

まず，最初は Top-down アルゴリズムを使用する状態から探索をスタートする．Top-down から Bottom-up への切り替えは，グラフに含まれる頂点数を $|V|$，辺の数を $|E|$，チューニングのためのパラメータを α および β，前のレベルで新たに到達した頂点の出次数の総和を m_f として，

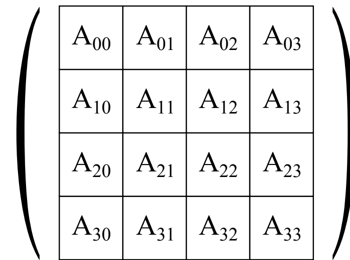


図 3 隣接行列の 2 次元分割

$m_f > |E|/\alpha$ となった場合に行われる．また，Bottom-up から Top-down への切り替えは，直前のレベルで到達した頂点の数を n_f として，$n_f < |V|^2/(\beta|E|)$ となった場合に行われる．なお，パラメータ α および β について，本稿の実装では先行研究 [2] で示された値である $\alpha = 10$，$\beta = 14$ の組み合わせを使用した．

2.4 クラスタシステムへの適用

クラスタシステムにおける幅優先探索においては，隣接行列の形式で表現されたグラフを 2 次元分割する方法 [6] が広く用いられている．2 次元分割による分割の例を**図 3** に示す．この方法では，全対全通信や Allreduce などの集団通信を同じ行を担当しているプロセス同士のみで済ませることができるようになり，ノード間通信のスケーラビリティを改善することができる．また，先に示した形の 2 次元分割ではレベルごとに部分行列のマッピングを転置する必要がある．ここで，対称な位置にある部分行列の組を同じプロセスが担当するようにすることで，マッピングの転置を前レベルからの入力を組になるブロック間でスワップさせる操作で済ませることができる [7].

3. GPU を用いた Hybrid-BFS

本稿で提案する手法では, 先に示したアルゴリズムにおける新たに訪問可能となる頂点の列挙する処理に GPU を利用することで処理を高速化している．ここでは，Top-down, Bottom-up それぞれのアプローチについてどのようなアルゴリズムを用いて GPU 上で処理を行っているかについて述べる．なお，いずれのアプローチにおいてもグラフの表現形式として隣接行列を CRS（Compressed Row Storage）形式で表現したものを用いている．

3.1 Top-down アプローチ

Top-down アプローチの GPU 実装には Two-phase 法 [8] を元に，ビットマップ更新処理にアトミック演算を用いるようにしたものを使用している．元の Two-phase 法ではビットマップの更新を厳密に行わず，必要に応じてラベルデータを参照することやワープ単位・スレッドブロック単位での重複排除を行うことでパフォーマンスを向上させているが，Hybrid-BFS を用いて直径の小さなグラフを処理

© 2013 Information Processing Society of Japan

する際に処理時間の多くを占める Bottom-up アプローチを効率よく実行するために正確に更新されたビットマップが必要となるため，このような対処をとっている．

また，シングルノード用の Two-phase 法では入力となる頂点の集合は配列として表現されたものとなっているが，本稿の実装ではノード間通信で得られる頂点集合の表現形式はビットマップとなっているため，ビットマップによる集合の表現から配列へ表現形式を変換する GPU カーネルを追加した．

3.2 Bottom-up アプローチ

Bottom-up アプローチの GPU 実装においては，訪問状態確認の対象となっている頂点の入次数によって処理方法を切り替えることとした．

まず，入次数が大きい頂点については同じワープに属するスレッドが同じ頂点についての訪問確認を行う．このアルゴリズムを**図 4** に示す．また，このアルゴリズムを用いた場合のグラフデータへのアクセスパターンを**図 5** に示す．このアルゴリズムは，十分な入次数を持った頂点を処理する際には Warp divergence の発生を抑えることができることや，辺の情報を取得するためのメモリアクセスをコアレスアクセスにすることができることなどから効率よく処理することができる．しかし，入次数がワープあたりのスレッド数に対して小さい頂点を処理する際には何も処理をしていないスレッドが増えてしまうため，その効率は大きく低下してしまう．

入次数が小さい頂点については，各スレッドがそれぞれ別の頂点についての訪問確認を行う．このアルゴリズムを**図 6** に示す．また，このアルゴリズムを用いた場合のグラフデータへのアクセスパターンを**図 7** に示す．このアルゴリズムでは，辺の情報を読み出すためのメモリアクセスがコアレスアクセスとならないものの，入次数の小さい頂点のみを処理している状況において処理を行っていないスレッドの数を減らすことができる．

各頂点にどちらのアルゴリズムを適用するかは，初期化時に入次数が閾値以上の頂点を示すビットマップを生成することで効率よく判定することができるようにしている．また，本稿の実装では最低でも 1 ループは全ワープを使い切るようにするため，アルゴリズム決定に用いる閾値はワープあたりのスレッド数と同じ値とした．

3.3 CPU と GPU の協調計算

幅優先探索は CPU による実装と GPU による実装の性能差がそれほど大きくないため，ホスト・デバイス間の通信コストを抑えることができれば CPU と GPU の協調計算による高速化を期待することができる．シングルノード実装においては，ある程度まで CPU のみで計算した後にそこまでの結果をデバイスメモリへ転送し，それ以降の処理をすべて GPU で行うといった提案もなされている [9]．

```
function warp-bottom-up (rows, cols, in, vis, next, preds)
  volatile shared selected[]
  bits ←
    not vis[cta_offset + thread_id] and
    large_degree_mask[cta_offset + thread_id]
  local_next ← 0
  while any(bits ≠ 0) do
    lower ← 31 - clz(bits)
    if bits ≠ 0 then
      selected[warp] ← thread_id
    end if
    mask ← 0
    if selected[warp] = thread_id then
      mask ← (1 << lower)
      bits ← bits and not mask
      selected[warp] ← (cta_offset + thread_id) × 32 + lower
    end if
    current ← rows[selected[warp]] + lane
    end ← rows[selected[warp] + 1]
    while current − lane < end do
      found ← current < end and cols[current] ∈ in
      if found then
        preds[selected[warp]] ← cols[current]
      end if
      if any(found) then
        local_next ← local_next or mask
        break
      end if
      current ← current + warp_size
    end while
  end while
  next[thread_id] ← local_next
```

図 4 次数が大きい頂点の処理

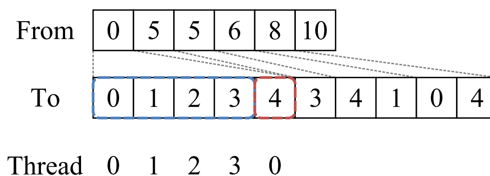


図 5 次数が大きい頂点の処理に用いるアルゴリズムのメモリアクセスパターン（warpSize = 4 の場合）

マルチノード実装では，RDMA for GPUDirect [10] などのノードをまたいだ GPU 間通信を効率よく行うための技術を利用しない場合には，GPU で計算した結果のほとんどはノード間通信のために一度ホストメモリへ転送されるため，ホスト・デバイス間の通信コストは GPU のみで計算処理を行った場合に比べてあまり大きくならない．このことを利用し，全レベルにおいて CPU と GPU で並列に探索処理を行うことで探索処理のさらなる高速化を図った．

3.4 計算と通信のオーバーラップ

幅優先探索における CPU による計算と MPI による通信

```
function thread-bottom-up (rows, cols, in, vis, next, preds)
  shared scan[], scratch[]
  bits ←
    not vis[cta_offset + thread_id] and
    not large_degree_mask[cta_offset + thread_id]
  cta_prefix_scan(scan, popcount(bits))
  position ← scan[thread_id]
  while bits ≠ 0 do
    shift ← 31 - clz(bits)
    scratch[position] ← (cta_offset + thread_id) × 32 + shift
    bits ← bits and not (1 << shift)
    ++position
  end while
  syncthreads()
  current ← thread_id
  end ← scan[block_dim]
  while current - lane < end do
    v ← scratch[current]
    scratch[current] ← 0
    for i ∈ [rows[v], rows[v + 1]) do
      if cols[i] ∈ in then
        preds[v] ← cols[i]
        scratch[current] ← (1 << (v and 31))
        break
      end if
    end for
    current ← current + block_dim
  end while
  syncthreads()
  local_next ← 0
  for i ∈ [scan[thread_id], scan[thread_id + 1]) do
    local_next ← local_next or scratch[i]
  end for
  next[thread_id] ← local_next
```

図 6 次数が小さい頂点の処理

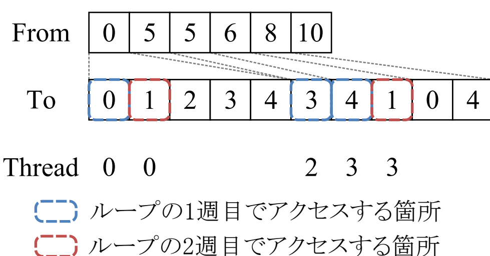


図 7 次数が小さい頂点の処理に用いるアルゴリズムのメモリアクセスパターン（warpSize = 4 の場合）

のオーバーラップはすでに提案されている [11] が，GPU を利用するようになるとさらに GPU による演算とホスト・デバイス間の通信について考慮する必要が生じる．本稿の実装では，**図 8** に示す流れで各レベルの処理を行う．

本稿の実装では，CPU と GPU による演算処理をオーバーラップさせているほか，ノード間通信を行っている裏で GPU によるデータ振り分けを行うことで，その後の CPU による演算結果と GPU による演算結果のマージをより効率よく行えるようにしている．

表 1 評価環境

CPU	Intel Xeon E5-2670 2.6GHz × 2
Main Memory	DDR3 1600MHz 128GB
GPU	NVIDIA Tesla M2090 × 4
GPU Memory	GDDR5 24GB（1GPU あたり 6GB）
Interconnection	InfiniBand x4 QDR × 2
Compiler	GCC 4.4.5 (-O2)
CUDA Toolkit	5.0
CUDA Compiler	nvcc release 5.0, V0.2.1221 (-gencode arch=compute_20,code=sm_20 -O2)
MPI	MVAPICH2 1.9b
計算ノード数	268

4. 性能評価

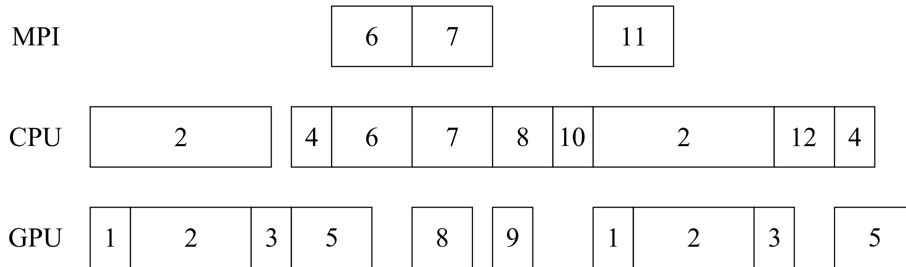
4.1 方法

評価環境には筑波大学計算科学研究センターにに設置されている HA-PACS を用いた．HA-PACS の諸元を**表 1** に示す．本稿における評価では，268 台の計算ノード中，最大 64 ノードまでの範囲で測定を行った．測定は，CPU のみを用いた実装（CPU only）と演算部分を GPU 化したもの（GPU only）と演算部分を CPU と GPU で協調計算するようにしたもの（CPU + GPU）の 3 つについて行った．なお，CPU+GPU のタスク分割比率は CPU:GPU=1:3 を使用した．この比率は CPU:GPU=0:1, CPU:GPU=1:3，CPU:GPU=1:1 の 3 パターンで計測したところ，CPU:GPU=1:3 の場合が最も良い性能を示したことによる．

性能評価に用いるグラフデータについては，Graph500 ベンチマークによって生成されるものを用いた．また，全体の探索性能を示す TEPS（Traversed Edges Per Second : 1 秒間にたどった辺の数）値の算出には Graph500 ベンチマークのものと同じ方法を用いている．この方法ではランダムに始点を 64 個選び，それぞれの場合の結果の中央値を測定結果としている．処理対象となるグラフのサイズは $|V| = 2^{23} \times$ ノード数，$|E| = |V| \times 16$ となるように設定した．

また，分割されたグラフの割り当てを単純にするためにプロセス数を 2^{2n+1} となるように設定している．このため，ノード数が 2^{2n} の場合と 2^{2n+1} の場合で 1 プロセスあたりのリソース配分が変化している．それぞれの場合の 1 プロセスあたりのリソース配分を**表 2** に示す．なお，1 ノードあたりのプロセス数はノード数が 2^{2n} の時は 2，ノード数が 2^{2n+1} の時は 4 となっている．ノード数によっては 1 プロセスに 2GPU が割り当てられているが，この場合の GPU 間のタスク分割は 1GPU に分割した隣接行列行列 1 ブロックを担当させることとした．また，各実装の CPU 処理部分は OpenMP を用いてプロセスに割り当てられたコア数と同数のスレッドを利用するように並列化した．

© 2013 Information Processing Society of Japan



1. 前レベルの入力をコピー
2. 到達可能になった頂点の列挙
3. 到達可能になった頂点集合のコピー
4. 到達可能になった頂点集合のマージ
5. 列挙した頂点を送信先ノードごとに振り分け
6. 親頂点情報をどのノードから送るかの割り当て
7. オールギャザによる到達状態の共有
8. 送信する必要のある親頂点情報の抽出
9. 抽出した親頂点情報のコピー
10. 抽出した親頂点情報のマージ
11. 親頂点情報の全対全通信による交換
12. 親頂点情報の書き込み

図 8 CPU+GPU 版 Level-synchronized BFS の流れ

表 2 1 プロセスに割り当てられるリソース

ノード数	CPU	GPU
2^{2n}	8 cores	2 GPUs
2^{2n+1}	4 cores	1 GPU

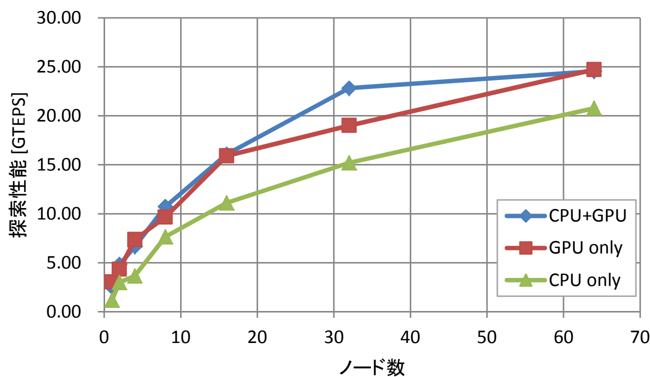


図 9 探索性能（HA-PACS）

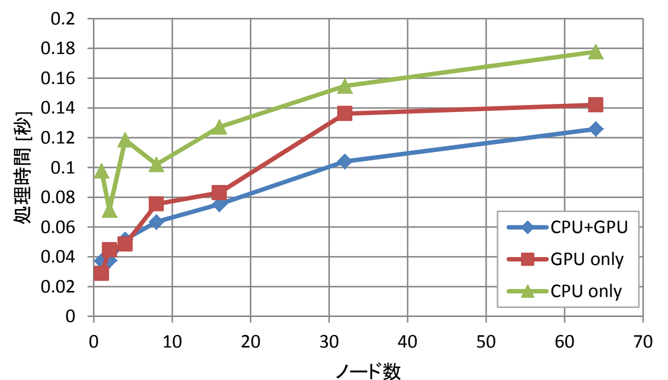


図 10 演算時間（HA-PACS）

4.2 結果

まず，全体の探索性能を計測した結果を**図 9** に示す．この結果より，GPU を利用することで CPU のみを利用した場合に比べて探索性能が向上していることが分かる．CPU と GPU の併用については，ノード数によって CPU を使用しないほうが高速なケースと併用したものの方が高速なケースの両方のパターンが現れた．

次に，探索処理のうち頂点集合と通信用データを生成するための演算のみに要した時間を計測した結果を**図 10** に示す．演算処理のみに注目した場合，GPU を利用することにより CPU のみを用いた場合に比べて約 1.4~3.3 倍の高速化を達成している．

5. 考察

まず，グラフ全体におけるある次数の頂点が占める頂点数の割合を**図 11** に示す．なお，この図は $|V| = 2^{22}$ として Graph500 ベンチマークによって生成されたグラフを元に作成したが，より大きな問題サイズであっても同様の傾向となる．GPU を用いた探索処理においては頂点の次数によって処理方法を選択しているが，今回の評価で用いた環境における閾値となる次数 32 を超える頂点は約 10%となっていた．また，クラスタシステム上で探索を行うために隣接行列を分割するが，分割された 1 ブロックに含まれる辺のみから求められる次数は平均すると $d/\sqrt{2P}$（d = グラフ全体における次数，P = プロセス数）となり，プロセス数を増やせば増やすほど次数の小さい頂点用のアルゴリズムで処理される頂点が増えることとなる．たとえば，$P = 128$ の場合にはブロック内での次数の平均が 32 を超える頂点は約 0.8%となる．これより，ノード数を増やしていくにつれて次数の小さい頂点を効率よく処理するためのアルゴリズムの重要度が増していくと考えられる．

次に，全体の探索性能のグラフを見ると，ノード数を増やすにつれて探索性能が飽和しつつある．しかし，演算に要した時間のみを見ると全体の探索性能ほど大きな性能飽和は見られないため，ノード間通信がスケーリングを妨げる要因となっていると考えられる．**図 12** に CPU と GPU を併用した実装におけるノード間通信に要した時間を示す．この図より，通信時間がノード数に対してほぼ線形に増加してしまっていることが分かる．2 次元分割を用いた幅優先探索ではノードあたりの通信量が $\sqrt{P}$ に比例するこ

© 2013 Information Processing Society of Japan

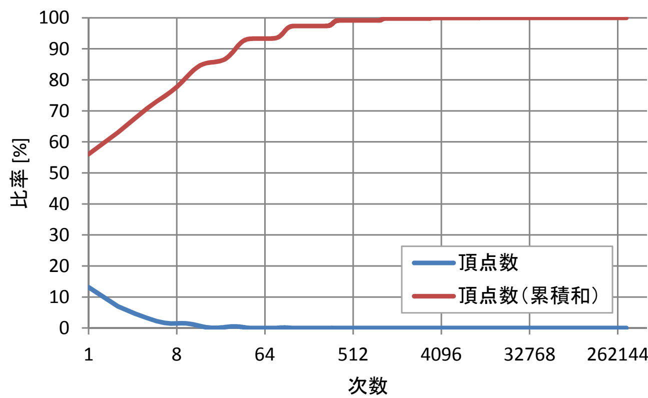


図 11 次数ごとの頂点出現頻度

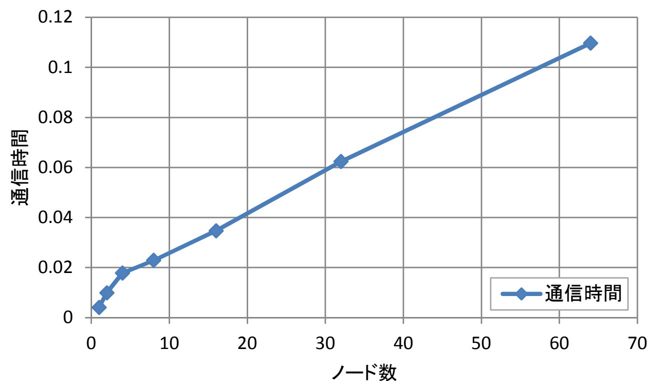


図 12 ノード間通信に要した時間（HA-PACS）

とと，いくつかの先行研究においてはさらに大きなノード数においても良好なスケーリング特性を示していることから，本稿で用いた実装が適切なものとなっていないことが原因と考えられる．

また，ノード数によって CPU を併用したほうが高速となるケースと GPU のみで計算したほうが高速なケースの両方が現れている．このことから，CPU と GPU の処理量の比率を問題やノード数に合わせて適切に選択することもより高速な探索につながると考えられる．

6. まとめ

本稿では，GPU を活用したクラスタシステムにおける幅優先探索アルゴリズムを実装し，それによって性能が向上することを示した．また，CPU と GPU を併用することで CPU および GPU いずれかのみの場合よりも高速な実装を実現した．

今後の課題として，まずスケールさせるうえでボトルネックとなっているノード間通信の改善が挙げられる．このための手法として，さらに効率的な通信と演算の多重化や通信内容の圧縮などの手法が提案されている [12].

また，CPU と GPU で協調して計算を行う実装においては CPU と GPU 間のタスク分割比率を設定しなければならないが，最適な比率はシステムの性能やノード数によって変化するため，効率的にパラメータを決定するための手法や自動チューニングのための手法について検討する必要がある．

参考文献

[1] Graph500: Brief Introduction | Graph 500, Graph500 (online), available from ⟨http://www.graph500.org/⟩ (accessed 2013-05-05).

[2] Beamer, S., Asanović, K. and Patterson, D.: Direction-optimizing breadth-first search, *Proc. 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, No. 12 (2012).

[3] Beamer, S., Buluc , A., Asanovi, K. and Patterson, D. A.: Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search, Technical Report UCB/EECS-2013-2, EECS Department, University of California, Berkeley (2013).

[4] Agarwal, V., Petrini, F., Pasetto, D. and Bader, D. A.: Scalable Graph Exploration on Multicore Processors, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pp. 1–11 (2010).

[5] Chhugani, J., Satish, N., Kim, C., Sewall, J. and Dubey, P.: Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency, *Proc. 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS 2012)*, pp. 378–389 (2012).

[6] Buluç, A. and Madduri, K.: Parallel breadth-first search on distributed memory systems, *Proc. 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, No. 65 (2011).

[7] Checconi, F., Petrini, F., Willcock, J., Lumsdaine, A., Choudhury, A. R. and Sabharwal, Y.: Breaking the speed and scalability barriers for graph exploration on distributed-memory machines, *Proc. 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, No. 13 (2012).

[8] Merrill, D., Garland, M. and Grimshaw, A.: High Performance and Scalable GPU Graph Traversal, Technical report, Department of Computer Science, University of Virginia (2011).

[9] Hong, S., Oguntebi, T. and Olukotun, K.: Efficient Parallel Graph Exploration on Multi-Core CPU and GPU, *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, pp. 78–88 (2011).

[10] NVIDIA Corporation: NVIDIA GPUDirect | NVIDIA Developer Zone, NVIDIA Corporation (online), available from ⟨https://developer.nvidia.com/gpudirect⟩ (accessed 2013-05-05).

[11] Ueno, K. and Suzumura, T.: Highly scalable graph search for the Graph500 benchmark, *Proc. 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*, pp. 149–160 (2012).

[12] Satish, N., Kim, C., Chhugani, J. and Dubey, P.: Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing, *Proc. 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, No. 14 (2012).

© 2013 Information Processing Society of Japan