Vol.2019-HPC-171 No.10
2019/9/20

Order/Degree問題のための重みなしグラフにおける全点対間最短経路アルゴリズムの並列化

中尾 昌広[1,a)]　村井 均[1]　佐藤 三久[1]

概要：大規模並列計算機システムのネットワークトポロジを設計することは，グラフ理論上の Order/Degree 問題として定義することができる．Order/Degree 問題を解く際には，グラフの特徴量の 1 つである全点対間最短経路（APSP：All Pairs Shortest Path）を高速に求める必要がある．そこで本稿では，幅優先探索による APSP アルゴリズム（BFS-APSP）と隣接行列による APSP アルゴリズム（ADJ-APSP）の並列化および比較を行う．なお，本稿では重みなしグラフを対象とする．それぞれの逐次アルゴリズムおよび OpenMP を用いたスレッド並列アルゴリズムにおいては，ADJ-APSP の方が性能が高いことを示した．しかしながら，MPI を用いた並列アルゴリズムにおいては，MPI の最大並列数は BFS-APSP の方が ADJ-APSP よりも多いため，BFS-APSP の方が性能が高くなる場合があることを示した．さらに，ADJ-APSP について GPU を用いた並列化を行うことにより，CPU よりも最大 16.53 倍の性能向上を達成した．また，対称性を持つグラフに対して各 APSP アルゴリズムを適用させることで，計算時間を大幅に短縮できることを示した．

1. はじめに

スーパーコンピュータやデータセンタなどで用いられる大規模並列計算機システムでは，多数の計算ノード同士がネットワークで相互接続されている．それらのシステムの性能を引き出すためには，計算ノード間のホップ数の直径と平均距離が小さくなるようにネットワークトポロジを設計することが重要である [1–4]．その直径と平均距離を求めるためには，計算ノード間の全点対間最短経路（APSP：All Pairs Shortest Path）を計算する必要がある．

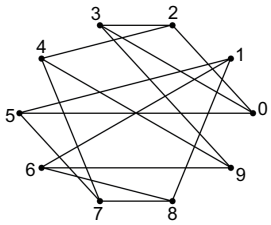
並列計算機システムの計算ノードを「頂点」，ネットワークの配線を「辺」とみなすことで，そのネットワークトポロジをグラフとして表現することができる．このことから，小さい直径と平均距離を持つネットワークトポロジを設計することは，グラフ理論上の Order/Degree 問題として定義できる [5]．Order/Degree 問題とは，与えられた頂点数（Order）と次数（Degree）を満たすグラフの集合の中から，最も小さい直径と平均距離を持つグラフを発見する問題である．本稿では計算ノード間のネットワークの性能はすべて同じであると仮定するため，重みなしグラフを取り扱う．Order/Degree 問題は，チップ内ネットワーク (Network on Chip) [6] などについても適用可能である．

Order/Degree 問題の定義は簡易であるが，効率的に解く手法は見つかっていない．そのため，焼き鈍し法（SA：Simulated Annealing）[7, 8] などのメタヒューリスティクスがよく用いられる [9–11]．しかしながら，メタヒューリスティクスは非常に多くの評価計算が必要になる．評価計算とは，現在の解に対して評価値を与えることであり，Order/Degree 問題では評価値として直径と平均距離がよく用いられる．幅優先探索（BFS：Breadth-First Search）による APSP アルゴリズム（BFS-APSP）の計算量は，頂点数 n と次数 d の場合は $O(n^2d)$ であるため，特に頂点数が増えるにつれ，APSP のための計算時間は大きくなる．

我々の過去の研究 [9] において，BFS-APSP に対して MPI と OpenMP を用いた並列化を行うことにより，計算時間の短縮を行った．本稿では，隣接行列による APSP アルゴリズム（ADJ-APSP）に対して並列化を行い，BFS-APSP との比較を行う．さらに，ADJ-APSP に対して GPU を用いた並列化も行う．なお，本稿で利用する全てのプログラムは `https://github.com/mnakao/APSP/` で公開している．

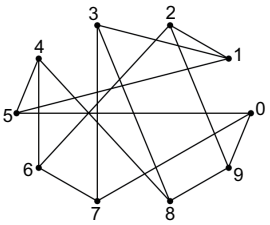
本稿の構成は下記の通りである．2 章では関連研究について述べ，3 章では BFS-APSP と ADJ-APSP の並列化について述べる．4 章では各 APSP アルゴリズムの性能評価を行い，5 章では ADJ-APSP における GPU を用いた並列化および対称性を持つグラフに対する各 APSP アルゴリズムの適用について述べる．6 章では本稿をまとめる．

[1] 理化学研究所 計算科学研究センター
RIKEN Center for Computational Science
[a)] masahiro.nakao@riken.jp



	0	1	2	3	4	5	6	7	8	9
0		2	1	1	2	1	3	2	3	2
1	2		3	3	3	1	1	2	1	2
2	1	3		1	1	2	3	2	3	2
3	1	3	1		2	2	2	3	3	1
4	2	3	1	2		2	2	1	2	1
5	1	1	2	2	2		2	1	2	3
6	3	1	3	2	2	2		2	1	1
7	2	2	2	3	1	1	2		1	2
8	3	1	3	3	2	2	1	1		2
9	2	2	2	1	1	3	1	2	2	

(a) 直径=3，平均距離=1.89



	0	1	2	3	4	5	6	7	8	9
0		2	2	2	2	1	2	1	2	1
1	2		1	1	2	1	2	2	2	2
2	2	1		2	2	2	1	2	2	1
3	2	1	2		2	2	2	1	1	2
4	2	2	2	2		1	1	2	1	2
5	1	1	2	2	1		2	2	2	2
6	2	2	1	2	1	2		1	2	2
7	1	2	2	1	2	2	1		2	2
8	2	2	2	1	1	2	2	2		1
9	1	2	1	2	2	2	2	2	1	

(b) 直径=2，平均距離=1.67

図 1: Example of Graphs $(n, d) = (10, 3)$ [9]

2. 関連研究

2.1 Order/Degree 問題

Order/Degree 問題とは，与えられた頂点数 n と次数 d を満たすグラフの直径と平均距離を最小化する問題のことである．$(n, d) = (10, 3)$ のグラフの例とそれぞれの距離行列を**図 1** に示す．距離行列はグラフの頂点間の距離を表している．図 1 は無向グラフであるため，距離行列は対称行列になる．距離行列の要素の最大値が直径であり，距離行列の要素の総和を要素数（$n^2 - n$）で割った値が平均距離である．**図 1a** よりも**図 1b** の方が直径と平均距離は小さいため，図 1b の方が良いグラフであるといえる．

Order/Degree 問題の国際コンペティション “Graph Golf” が国立情報学研究所の主催で開催されている [10]．Graph Golf は 2015 年から毎年開催されており，年ごとに異なる頂点数と次数の組合せが出題される．2019 年の出題は，$(n, d) = (50, 4)$，$(512, 4)$，$(512, 6)$，$(1024, 4)$，$(1726, 30)$，$(4855, 15)$，$(9344, 6)$，$(65536, 6)$，$(100000, 8)$，$(1000000, 16)$，$(1000000, 32)$ の 11 種類である．なお，頂点数は固定であるが，次数の値は最大値であり，指定された値より小さくても構わない[*1]．Graph Golf の参加者は，設定された期間内（2019 年は 5 月 13 日から 10 月 14 日）に公式サイトの Web フォームから自身が発見したグラフを投稿する．

Graph Golf の公式サイト [10] において，過去の参加者が開発した手法のスライドと論文が公開されている [11–13]．これまでの手法は，下記の 2 つに区分することができる．(1) SA などのメタヒューリスティクスを用いる．この手法は任意の頂点数と次数を持つグラフを作成できるという利点はあるが，膨大な評価計算が必要であるという問題点がある．この問題点を克服するため，APSP の算出に近似を用いる場合がある．しかし，近似が持つ誤差がメタヒューリスティクスの解探索性能に悪影響を与える可能性がある．そのため，我々は APSP を正確に求めることが重要であると考えている．(2) 頂点数が小さい複数のグラフを組合せる．この手法は APSP の計算回数が (1) と比較して少ないという利点はあるが，特定の頂点数と次数を持つグラフしか作れないという問題点がある．また，頂点数が多いグラフを作成する場合は，その組合せ数も膨大になるため，(2) においても APSP の計算時間の削減は重要である．

[*1] 例えば，(4855, 15) は頂点数と次数が共に奇数であるため，握手の補題により全頂点が 15 本の辺を持つグラフは存在しない．

2.2 全点対間最短経路アルゴリズム

APSP は重要なネットワーク特徴量の 1 つであるため，様々な APSP アルゴリズムが提案されているが，その多くは重みありグラフを対象としている．グラフの重みが均一であると仮定すると，重みありグラフを対象とした APSP アルゴリズムを重みなしグラフでも用いることができるが，その計算効率は低い．例えば，重みありグラフを対象とした APSP アルゴリズムの中で最も有名なワーシャルフロイド法 [14,15] の計算量は $O(n^3)$ である．それに対して，BFS-APSP の計算量は $O(n^2 d)$ である．Order/Degree 問題におけるグラフは，二重辺やループを含まない単純グラフであるため，常に $n > d$ が成り立つ．すなわち，$O(n^3) > O(n^2 d)$ であるため，BFS-APSP の方がワーシャルフロイド法よりも計算効率は高い．

重みなし無向グラフを対象とした効率的な APSP アルゴリズムに Seidel 法があり，その計算量は $O(n^{2.376} \log n)$ である [16]．すなわち，$n^{0.376} \log n < d$ の場合（頂点数に対して辺数が比較的多い密なグラフの場合），Seidel 法の方が BFS-APSP よりも計算効率は高くなる．しかしながら，本稿で対象とする Order/Degree 問題は，工業製品などへの利用を想定しているため，その次数は比較的小さいという特徴がある．例えば，2.1 節で述べた 2019 年の Graph Golf の出題は，すべて $n^{0.376} \log n > d$ が成り立つ．そのため，本稿では Seidel 法は扱わないこととする．

3. 全点対間最短経路アルゴリズムの並列化

本章では，BFS-APSP と ADJ-APSP の逐次アルゴリズムについて説明した後，MPI と OpenMP を用いた並列アルゴリズムについて説明する．

3.1 幅優先探索によるアルゴリズム

3.1.1 概要

BFS を用いると，ある頂点から他の頂点までの距離を求めることができる．そのため，全頂点から BFS を行うことにより，APSP を求めることができる．1 回の BFS の計算量は辺数に比例するため，$O(nd)$ である．それを n 回繰り返すため，BFS-APSP の計算量は $O(n^2 d)$ になる．

```
1  function SERIAL_BFS_APSP(vertices, nodes)
2    for source ∈ nodes
3      distances ← BFS(vertices, source)
4      diameter[source] ← MAX(distances)
5      distance[source] ← AVERAGE(distances)
6    max_diameter ← MAX(diameter)
7    average_distance ← AVERAGE(distance)
8    return max_diameter, average_distance
9
10 function BFS(vertices, source)
11   frontier ← {source}
12   next ← {}
13   distance ← {−1, −1, ..., −1}
14   k ← 1
15   while frontier ≠ {}
16     TOPDOWN(vertices, frontier, next, distances, k++)
17     frontier ← next
18     next ← {}
19   return distances
20
21 function TOPDOWN(vertices, frontier, next, distances, k)
22   for v ∈ frontier
23     for n ∈ neighbors(v, vertices)
24       if distances[n] = −1 then
25         distances[n] ← k
26         next ← next ∪ {n}
```

図 2: Serial BFS-APSP

3.1.2 逐次アルゴリズム

BFS-APSP の逐次アルゴリズムの擬似コードを**図 2** に示す．BFS には，Top-down アプローチ [17] を採用した．図 2 の 2-5 行目のループ文では，ある頂点を出発点とする BFS を頂点数だけ繰り返すことを示している．6-7 行目では，BFS によって得られた距離情報から，グラフの直径と平均距離を計算している．関数 `BFS` と `TOPDOWN` については，文献 [17] とほぼ同じであるため，説明を省略する．また，図 2 では省略しているが，キャッシュヒット率を上げるためにビットマップ [18] による最適化も行っている．

補足として，文献 [17] では，ソーシャルネットワークなどでよく現れるパターンを持つ Kronecker Graph [19] を対象とした Hybrid アプローチが提案されている．Kronecker Graph では，次数の大きい頂点と小さい頂点が混在しており，そのようなグラフでは，Top-down アプローチよりも Hybrid アプローチの方が高速に BFS を行うことができる．しかしながら，本稿で対象とするグラフは，すべての次数がほぼ同じであり，その次数も比較的小さい．予備実験を行った結果，Hybrid アプローチよりも Top-down アプローチの方が性能が高かったため，本稿では Top-down アプローチを用いた．

3.1.3 MPI と OpenMP による並列アルゴリズム

並列化の概要として，MPI を用いて複数の BFS を同時に行い，さらに OpenMP を用いて 1 つの BFS をスレッド分割する．なお，MPI および OpenMP の最大並列数は，共に n である．

```
1  function PARALLEL_BFS_APSP(vertices, nodes)
2    for source ∈ nodes on each process
3      distances ← BFS(vertices, source)
4      diameter[source] ← MAX(distances)
5      distance[source] ← AVERAGE(distances)
6    max_diameter ← ALLREDUCE(diameter, MAX)
7    average_distance ← ALLREDUCE(distance, SUM)
8    average_distance ← average_distance/nodes
9    return max_diameter, average_distance
10
11 function BFS(vertices, source)
12   frontier ← {source}
13   next ← {}
14   distance ← {−1, −1, ..., −1}
15   k ← 1
16   while frontier ≠ {}
17     TOPDOWN(vertices, frontier, next, distances, k++)
18     frontier ← next
19     next ← {}
20   return distances
21
22 function TOPDOWN(vertices, frontier, next, distances, k)
23   count ← 0
24   for v ∈ frontier omp parallel
25     local_next ← {}
26     local_count ← 0
27     for n ∈ neighbors(v, vertices)
28       if distances[n] = −1 then
29         distances[n] ← k
30         local_next ← local_next ∪ {n}
31         local_count++
32       omp critical
33         next[count] ← local_next
34         count ← count+local_count
```

図 3: Parallel BFS-APSP

BFS-APSP の並列アルゴリズムの擬似コードを**図 3** に示す．2-5 行目では，出発点を各プロセスに割り振り，BFS を並列実行している．6-8 行目では，集合通信を用いて各プロセスが持つ距離情報を集約し，直径と平均距離を計算している．関数 BFS は，図 2 と同じである．24-34 行目では，頂点集合 $frontier$ の探索を各スレッドが分割して行っている．27-34 行目では，各スレッドは新しく発見した探索点をプライベート変数 *local_next* に保存し，スレッドに対する排他制御を用いて共有変数 *next* を *local_next* から作成している．

3.2 隣接行列によるアルゴリズム

3.2.1 概要

あるグラフの隣接行列を A とし，さらに A の対角要素を 1 にセットする．A^k のある要素 $a_{i,j}$ の値が 0 でない場合，頂点 i から頂点 j までは，k ホップ以内に到達できることを表す．この性質を利用することで，グラフの直径と平均距離を求めることができる．k を 1 から 1 つずつ順に増やしていき，全要素の値が 0 でなくなった時点の k の値は，そのグラフの直径になる．また，k を 1 から直径まで順に増やす度に，A^k における要素 $a_{i,j}$ の値が 0 である要素の数

```
1  function SERIAL_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← nodes*(nodes−1)
4    elements ← ⌈nodes/E⌉
5    A, B ← INITIALIZE(nodes, elements)
6    for k=1 ... nodes−1
7      for i=1 ... nodes
8        for n ∈ neighbors(i, vertices)
9          for j=1 ... elements
10           B[i][j] ← B[i][j] | A[n][j]
11
12     num ← 0
13     for i=1 ... nodes
14       for j=1 ... elements
15         num ← num+POPCNT(B[i][j])
16
17     if(num = nodes*nodes) break
18
19     SWAP(A, B)
20     diameter++
21     distance ← distance+(nodes*nodes−num)
22   average_distance ← distance/((nodes−1)*nodes)
23   return diameter, average_distance
```

図 4: Serial ADJ-APSP

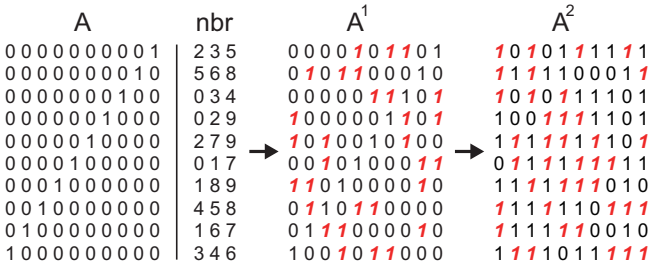


図 5: Example of ADJ-APSP

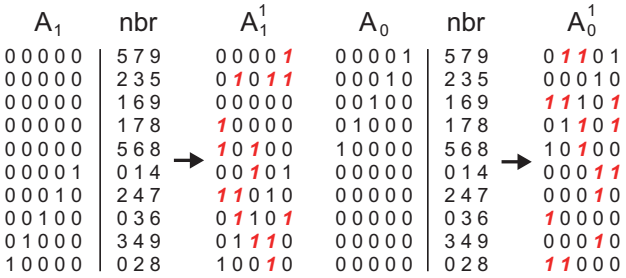


図 6: Example of Divided ADJ-APSP

(k ホップ目における未到達の頂点数）を足すことにより，各頂点間の距離の合計値を求めることができる．その合計値を (n^2-n) で割った値が平均距離になる．このアルゴリズムの計算量は，直径を D，隣接行列の 1 要素のビット数を E とした場合，$O(n^2dD/E)$ である．Order/Degree 問題で扱うグラフの直径は比較的小さいという特徴がある．例えば，2019 年の Graph Golf の出題の直径の理論的な下界は，すべて 10 未満である．また，実装では隣接行列の型に `uint64_t` を用いているため，$E = 64$ である．

3.2.2 逐次アルゴリズム

ADJ-APSP の逐次アルゴリズムの実装は [20] を参考にした．その逐次アルゴリズムの擬似コードを**図 4** に示す．まず，$n \times n$ ビットの隣接行列 A と B を用意する．4 行目では，隣接行列の列の要素数 *elements* を計算している．5 行目では，隣接行列に対して，各行の右から n ビット目を 1 に，それ以外を 0 に初期化する．6 行目では，A^k を求めるため，k を 1 からインクリメントしている．k の最大値が $(nodes-1)$ である理由は，グラフの直径は $(n-1)$ を上回ることはないからである．7-10 行目では，A^k を計算するため，隣接リスト *neighbors* を介して論理和を計算し，その結果を B に代入している．12-15 行目では，隣接行列 B の 1 のビット数（k ホップ目で到達した頂点数）である *num* を計上している．21 行目で隣接行列の全要素数 $nodes * nodes$ から *num* を引くことにより，0 のビット数を計上する．関数 `POPCNT` は，1 要素に含まれる 1 のビットを数える関数であり，実装ではアーキテクチャによって `__builtin_popcountll` と `_mm_popcnt_u64` のどちらかを用いている．17 行目では，全要素が 1 になったとき（全頂点に到達したとき），ループから抜け出している．19 行目では，次のイテレーションのために，A と B を入れ替えている．20-22 行目では，直径と平均距離を計算している．

図 5 に，図 1a を用いた本アルゴリズムの具体例を示す．実装の都合上，図 1a の距離行列と図 5 の行列 A^k とは左右反転している．図 5 の左側の行列 A は，初期化した状態を表している．隣接リスト *nbr*（図 4 の 8 行目で利用している *neighbors* と同じ）の i 行目には，頂点番号 *i-1* と隣接している頂点番号が格納されている．A から A^1 の変更点は赤太字で示しており，その箇所は図 1a の距離行列の “1” に該当する箇所である．A^1 から A^2 の変更点も赤太字で示しており，その箇所は図 1a の距離行列の “2” に該当する箇所である．なお，A^3 のすべてのビットは 1 である．

隣接行列のサイズは $(n^2/8)$ Byte であるため，頂点数 n の 2 乗に比例して，利用メモリ量は増える．例えば，2019 年の Graph Golf の出題の中の最大頂点数は $n = 1,000,000$ であり，その隣接行列のサイズは約 116GByte である．そのため，図 4 のアルゴリズムでは大規模なグラフの計算をラップトップなどで行うことは難しい．そこで，隣接行列を縦に分割し，それぞれ独立して計算を行う手法（以下，Divided ADJ-APSP）について述べる．Divided ADJ-APSP を図 5 に対して適用した例を**図 6** に示す．図 6 では，隣接行列 A を 2 分割している．A^k の上位 5 ビットは A_1^k であり，A^k の下位 5 ビットは A_0^k である．分割数を *parsize* とすると，隣接行列のサイズは $(n^2/8/parsize)$ Byte になる．

Divided ADJ-APSP の擬似コードを**図 7** に示す．図 7 の 5 行目で利用されている定数 *CHUNK* は，分割された隣接行列の列の要素数である．6 行目では，隣接行列を分割した数だけ，以下の処理を繰り返すことを示している．7 行

```
1  function SERIAL_DIVIDED_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← nodes∗(nodes−1)
4    elements ← ⌈nodes/E⌉
5    parsize ← ⌈elements/CHUNK⌉
6    for c=1 ... parsize
7      A, B ← INITIALIZE(nodes, CHUNK)
8      for k=1 ... nodes−1
9        for i=1 ... nodes
10         for n ∈ neighbors(i, vertices)
11           for j=1 ... CHUNK
12             B[i][j] ← B[i][j] | A[n][j]
13
14       num ← 0
15       for i=1 ... nodes
16         for j=1 ... CHUNK
17           num ← num+POPCNT(B[i][j])
18
19       if(num = nodes∗CHUNK∗E) break
20
21       SWAP(A, B)
22       distance ← distance+(nodes∗CHUNK∗E−num)
23     diameter ← MAX(diameter, k+1)
24   average_distance ← distance/((nodes−1)∗nodes)
25   return diameter, average_distance
```

図 7: Serial Divided ADJ-APSP

目以降は，変数 *elements* の代わりに定数 *CHUNK* を用いている以外は図 4 とほぼ同じであるため，説明は省略する．また，図 7 では省略しているが，*nodes* が $CHUNK$ で割り切れなかった場合の対応も行っている．なお，12 行目の論理和において，SIMD とメモリバンド幅を有効利用するためには，$(CHUNK \times E)$ の値は 512 の倍数かつある程度大きな値が望ましい．そのため，実装では $CHUNK$=64 に設定している．

予備実験において，ADJ-APSP と Divided ADJ-APSP との性能を比較した結果，ADJ-APSP の方が僅かに性能が高いことを確かめている．そのため，実装では，大きなグラフを扱う場合のみ，自動的に Divided ADJ-APSP を用いることにしている．具体的には隣接行列のサイズ $(n^2/8)$ の値が 2^{31} を上回ったときに，Divided ADJ-APSP を実行している．

3.2.3 MPI と OpenMP による並列アルゴリズム

Divided ADJ-APSP と同様の分割方法を用いると，MPI を用いて ADJ-APSP の並列化が可能である．また，ADJ-APSP に存在するいくつかのループ文はデータ依存性がないため，OpenMP によるスレッド並列化も行うことができる．なお，MPI の最大並列数は隣接行列の列の要素数と同じ $\lceil n/E \rceil$ であり，OpenMP の最大並列数は n である．

ADJ-APSP の並列アルゴリズムの擬似コードを**図 8** に示す．5 行目の変数 *procs* は MPI のプロセス数であり，変数 *chunk* は各プロセスが処理する隣接行列の列の要素数である．7 行目では，ループ文のイテレーションを各プロセスに割り振り，並列実行している．8-24 行目は図 7 とほぼ同じであるが，定数 *CHUNK* が変数 *chunk* に置き換わっている．また，10 と 16 行目では OpenMP によるスレッド並列を行っている．25-27 行目では，集合通信を用

```
1  function PARALLEL_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← 0
4    elements ← ⌈nodes/E⌉
5    chunk ← ⌈elements/procs⌉
6    parsize ← ⌈elements/chunk⌉
7    for c=1 ... parsize on each process
8      A, B ← INITIALIZE(nodes, chunk)
9      for k=1 .. nodes−1
10       for i=1 ... nodes omp parallel
11         for n ∈ neighbors(i, vertices)
12           for j=1 ... chunk
13             B[i][j] ← B[i][j] | A[n][j]
14
15       num ← 0
16       for i=1 ... nodes omp parallel reduction(+:num)
17         for j=1 ... chunk
18           num ← num+POPCNT(B[i][j])
19
20       if(num = nodes∗chunk∗E) break
21
22       SWAP(A, B)
23       distance ← distance+(nodes∗chunk∗E−num)
24     diameter ← MAX(diameter, k+1)
25   diameter ← ALLREDUCE(diameter, MAX)
26   average_distance ← ALLREDUCE(distance, SUM)
27   average_distance ← average_distance/((nodes−1)∗nodes)+1
28   return diameter, average_distance
```

図 8: Parallel ADJ-APSP

```
1  function PARALLEL_DIVIDED_ADJ_APSP(vertices, nodes)
2    diameter ← 1
3    distance ← 0
4    elements ← ⌈nodes/E⌉
5    parsize ← ⌈elements/CHUNK⌉
6    for c=1 ... parsize on each process
7      A, B ← INITIALIZE(nodes, CHUNK)
8      for k=1 .. nodes−1
9        for i=1 ... nodes omp parallel
10         for n ∈ neighbors(i, vertices)
11           for j=1 ... CHUNK
12             B[i][j] ← B[i][j] | A[n][j]
13
14       num ← 0
15       for i=1 ... nodes omp parallel reduction(+:num)
16         for j=1 ... CHUNK
17           num ← num+POPCNT(B[i][j])
18
19       if(num = nodes∗CHUNK∗E) break
20
21       SWAP(A, B)
22       distance ← distance+(nodes∗CHUNK∗E−num)
23     diameter ← MAX(diameter, k+1)
24   diameter ← ALLREDUCE(diameter, MAX)
25   average_distance ← ALLREDUCE(distance, SUM)
26   average_distance ← average_distance/((nodes−1)∗nodes)+1
27   return diameter, average_distance
```

図 9: Parallel Divided ADJ-APSP

表 1: Specification of the K computer

CPU	SPARC64 VIIIfx (8Cores, 2.0GHz)
Memory	DDR3 (64GB/s, 16GB)
Network	Torus fusion six-dimensional mesh/torus network, 5GB/s × 10
Software	Fujitsu Compiler K-1.2.0-25

表 2: Specification of the Cygnus system

CPU	Intel Xeon Gold 6126 (12Cores, 2.6GHz) × 2
Memory	DDR4 (128GB/s × 2, 192GB)
GPU	NVIDIA Tesla V100 (900GB/s, 32GB) × 4
Network	InfiniBand HDR100 (12.5GB/s) × 4
Software	intel/19.0.3, mvapich/2.3.1, cuda/10.1

いて各プロセスが持つ情報を集約し，直径と平均距離を計算している．なお，図 7 とは異なり，図 8 の 3 行目で変数 *distance* の初期値を 0 としている理由は，26 行目で各プロセスが持つ変数 *distance* に対する集約演算を行うからである．実際の初期値 $(nodes * (nodes - 1))$ は，27 行目で *average_distance* に対して 1 を足すことで同等の処理を行っている．

Divided ADJ-APSP も MPI と OpenMP を用いた並列化が可能である．その疑似コードを**図 9** に示す．図 9 と図 8 との違いは，図 8 中の変数 *chunk* の代わりに定数 *CHUNK* を用いただけである．

4. 評価

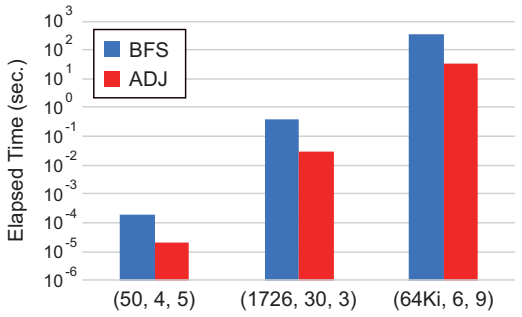
4.1 実験環境

本章では，3 章で述べた 2 種類の APSP アルゴリズムの性能比較を行う．実験には，理化学研究所のスーパーコンピュータ「京」と筑波大学の Cygnus を用いた．それぞれの仕様を**表 1** と**表 2** に示す．評価に用いるグラフは，GraphGolf の公式サイト [10] で提供されているランダムグラフを利用した．

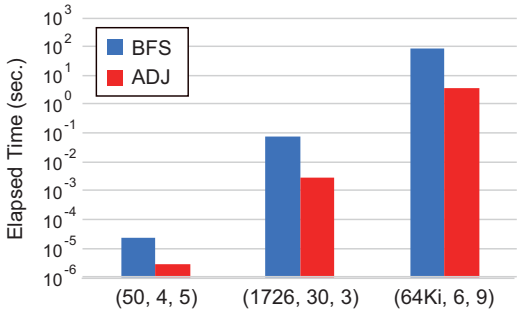
4.2 逐次アルゴリズムの性能評価

3.1.2 節と 3.2.2 節で述べた逐次の各 APSP アルゴリズムの性能評価を行う．評価に用いるグラフは，$(n, d, D) = (50, 4, 5)$，$(1726, 30, 3)$，$(65536, 6, 9)$ である（以下，$(65536, 6, 9)$ は $(64Ki, 6, 9)$ と記述する）．なお，隣接行列による APSP アルゴリズムでは，すべてのグラフにおいて，Divided ADJ-APSP ではなく ADJ-APSP で実行している．

結果を**図 10** に示す．この結果より，すべての場合において，ADJ-APSP の方が BFS-APSP よりも高速であることがわかる．「京」は 9.65〜13.38 倍，Cygnus は 8.08〜29.49 倍の速度差であった．また，Cygnus の方が「京」よりも高速であり，BFS-APSP は 4.14〜8.32 倍，ADJ-APSP は 6.96〜10.40 倍の速度差であった．



(a) On the K computer



(b) On the Cygnus system

図 10: Results of Serial APSP Algorithm

4.3 並列アルゴリズムの性能評価

3.1.3 節と 3.2.3 節で述べた並列化した各 APSP アルゴリズムの性能評価を行う．評価に用いたグラフは，$(n, d, D) = (64Ki, 6, 9)$ と $(1000000, 32, 5)$ である（以下，$(1000000, 32, 5)$ は $(1M, 32, 5)$ と記述する）．

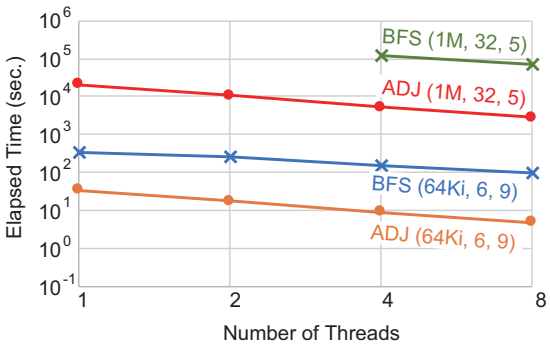
4.3.1 OpenMP による並列アルゴリズム

スレッド並列化の効果を調べるため，プロセス数を 1 に固定し，スレッド数を変化させて実行した．ただし，$(1M, 32, 5)$ を用いた BFS-APSP の結果の一部は，計算機のジョブの制限時間を越えてしまったため取得できなかった．また，$(1M, 32, 5)$ においては，ADJ-APSP ではなく Divided ADJ-APSP で実行している．

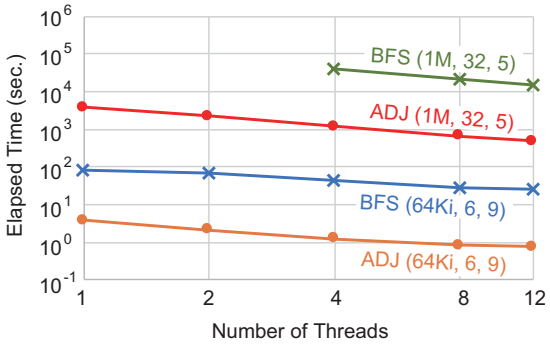
結果を**図 11** に示す．この結果より，同じ問題では ADJ-APSP の方が BFS-APSP よりも高速であることがわかる．次に，図 11 における速度向上率を**図 12** に示す．ただし，1 スレッドとの比較であるため，$(1M, 32, 5)$ の BFS-APSP については記載していない．この結果より，ADJ-APSP は BFS-APSP よりも速度向上率が高いことがわかる．この理由は，BFS-APSP はスレッドに対する排他制御が必要であるのに対し，ADJ-APSP はそのような制御は行っていないからである．

4.3.2 MPI と OpenMP による並列アルゴリズム

次に，スレッド数を各計算機環境の最大値（「京」は 8，Cygnus は 12）に設定し，プロセス数を変化させて各 APSP アルゴリズムを実行した．各 CPU に 1 プロセスを割り当

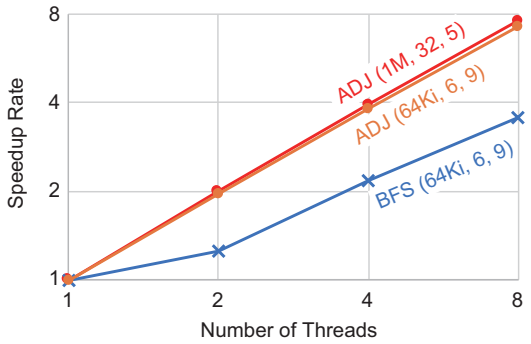


(a) On the K computer

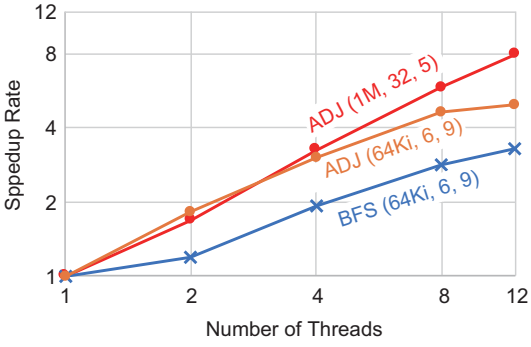


(b) On the Cygnus system

図 11: Results of Threaded APSP Algorithm



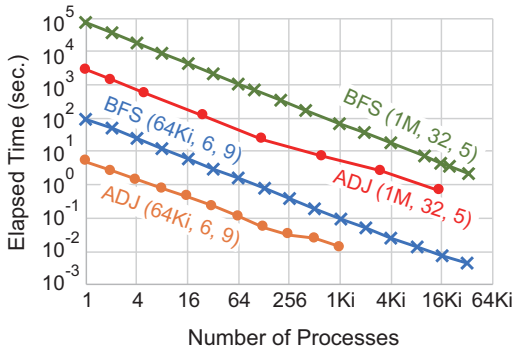
(a) On the K computer



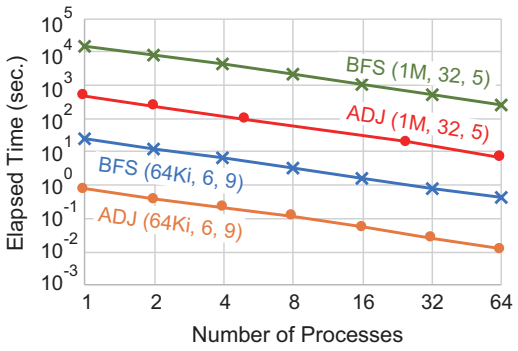
(b) On the Cygnus system

図 12: Speedup Rate for Threaded APSP Algorithm

てている．なお，(64Ki, 6, 9) と (1M, 32, 5) における MPI の最大並列数は，BFS-APSP は 65,536 と 1,000,000 であり，ADJ-APSP は 1,024 と 15,625 である．



(a) On the K computer



(b) On the Cygnus system

図 13: Results of Parallel APSP Algorithm

結果を**図 13** に示す．なお，(1M, 32, 5) では 25 プロセスまでは Divided ADJ-APSP で実行しており，それ以降は通常の ADJ-APSP で実行している．この結果より，同じ並列数の場合は ADJ-APSP の方が BFS-APSP よりも高速であることがわかる．しかしながら，BFS-APSP の方が最大並列数は大きいため，**図 13a** の (64Ki, 6, 9) においては，BFS-APSP の方が最大プロセス（1,024 プロセス）時の ADJ-APSP よりも高速になる場合があることがわかる．また，図 13a から 256 プロセスあたりから ADJ-APSP の並列化効率が徐々に低くなることがわかる．この理由は，プロセス数が増えるにつれ，隣接行列の列の要素数が少なくなるため，論理和に対する計算効率が低くなるからと考えられる．

5. 発展課題

5.1 GPU を用いた並列化

ADJ-APSP において最も時間を要する箇所は隣接行列の行に対する論理和（例えば，図 8 の 10-13 行目）である．この処理時間はメモリバンド幅に律速するため，GPU のような高速メモリを持つアクセラレータと相性が良いと考えられる．そこで，本節では並列アルゴリズムの ADJ-APSP ベースに，GPU 上で動作するように拡張を行う．

CUDA を用いた論理和を行うコードを**図 14** に示す．コアレスアクセスになるように，連番のスレッドが隣接行列の各行の論理和を計算するように実装している．ADJ-APSP

```
1  int tid = threadIdx.x + blockIdx.x * blockDim.x;
2
3  while (tid < nodes*elements) {
4    int i = tid / elements;
5    int k = tid % elements;
6    uint64_t tmp = B[i][k];
7    for(int j=0;j<num_degrees[i];j++){
8      int n = neighbors[i][j];
9      tmp |= A[n][k];
10   }
11   B[i][k] = tmp;
12   tid += blockDim.x * gridDim.x;
13 }
```

図 14: Code of Logical Sum in CUDA

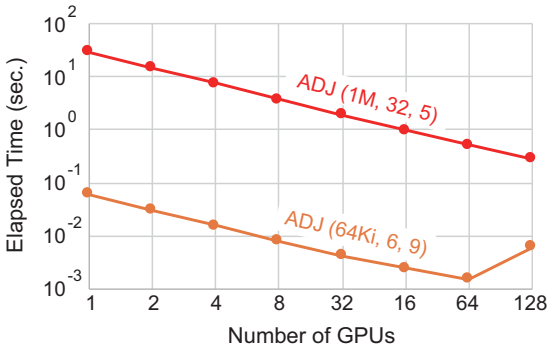


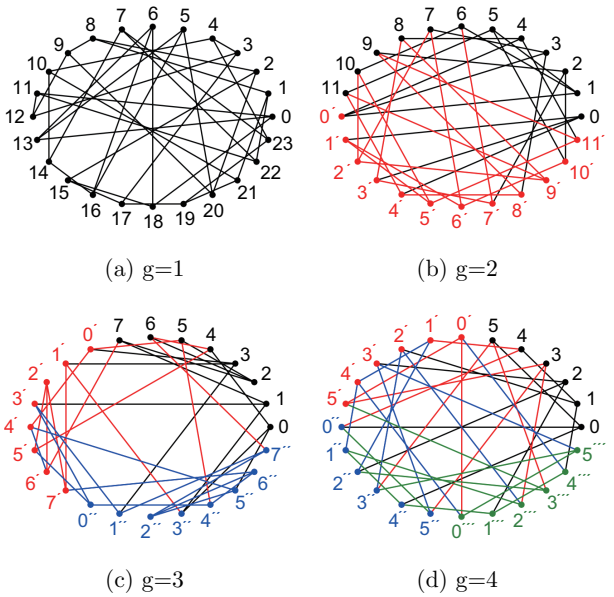
図 15: Results of ADJ-APSP using GPUs

の実装の他の箇所については簡易であるため省略するが，関数 POPCNT については，GPU メモリ上の隣接行列に対して処理を行うために，CUDA Math API の__popcll を用いている．

次に性能評価を行う．評価に用いるグラフは，$(n, d, D) = (64\text{Ki}, 6, 9)$ と $(1\text{M}, 32, 5)$ である．各プロセスは 1 台の GPU を担当するため，Cygnus の各計算ノードに最大 4 プロセスを割り当てた．結果を**図 15** に示す．この結果より，$(64\text{Ki}, 6, 9)$ の 128 プロセス時以外は，GPU 数が増えるにつれ計算時間が削減することがわかる．1GPU を用いたときの経過時間は，$(64\text{Ki}, 6, 9)$ は 6.01×10^{-2} 秒，$(1\text{M}, 32, 5)$ は 2.87 $\times10$ 秒である．これらの値は，**図 13b** の 1CPU 利用時の 12.47 倍と 16.53 倍の性能である．また，各グラフの最良値は，$(64\text{Ki}, 6, 9)$ は 64GPU 利用時で 1.59×10^{-3} 秒，$(1\text{M}, 32, 5)$ は 128GPU 利用時で 2.84×10^{-1} 秒である．すなわち，1GPU 利用時と比較して $(64\text{Ki}, 6, 9)$ は 38.41 倍，$(1\text{M}, 32, 5)$ は 101.10 倍の高速化を達成した．$(64\text{Ki}, 6, 9)$ の 128 プロセス時で性能が低下した理由は，この場合の隣接行列の列の要素数は 8 であるため，コアレスアクセスが発生する条件（16 スレッドがアクセスするアドレスがスレッド番号順に隣接）から外れたからである．

5.2 対称性を持つグラフに対する APSP アルゴリズム

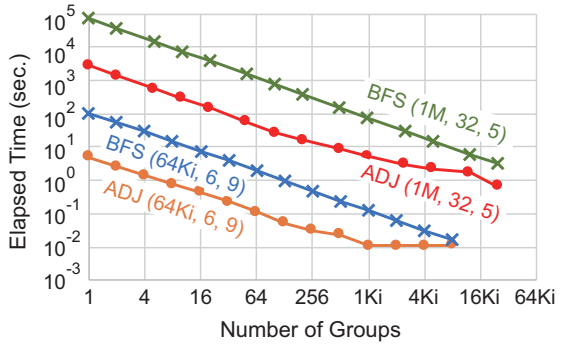
我々の過去の研究 [9] において，グラフに対称性を持たせることにより，BFS-APSP の計算量を削減し，かつ Order/Degree 問題においてメタヒューリスティクスの性能が向上することを示している．$(n, d) = (24, 3)$ の対称性を持つグラフの例を**図 16** に示す．図中の変数 g はグループ数であり，各グラフを平面として見た場合，$360/g$ 度回転させると頂点と辺の接続関係が同じグラフになる．そのため，グループ数 g は頂点数 n の約数である必要がある．なお，$g = 1$ のときは，対称性を持たない通常のグラフである．対称性を持たせたグラフの作成方法は下記の通りである．(1) 頂点数 n/g，次数 d の小グラフを作成する．(2) 小グラフを g 個複製する．(3) 各小グラフから適当な辺を選択し，他のグラフと接続する．ここで (1) で作成する小グラフの制約上，$n/g > d$ を満たす必要がある．



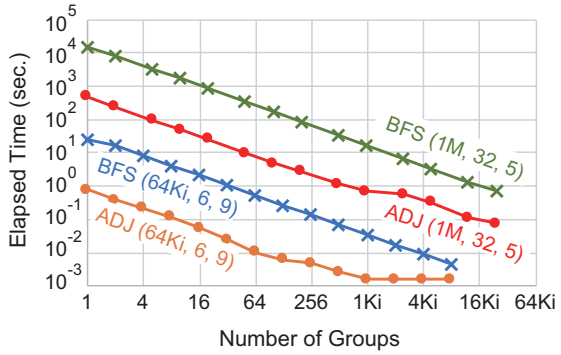
(a) g=1　(b) g=2　(c) g=3　(d) g=4

図 16: Example of Symmetrical Graphs $(n, d) = (24, 3)$ [9]

グラフの対称性を利用することにより，BFS-APSP の計算量を削減できる．対称性を持つグラフでは，対称的な関係のある頂点同士において，その頂点から他の頂点までの距離はすべて同じであるという性質がある．例えば，**図 16d** では，頂点 0，0′，0″，0‴ から，他の頂点までの距離の集合はすべて同じであるため，頂点 0〜5 から BFS を行うだけでグラフ全体の APSP を求めることができる．すなわち，グループ数 g のグラフにおいて BFS-APSP を用いて APSP を求めるには，(n/g) 個の頂点から BFS を行うだけでよい．ADJ-APSP においても，BFS-APSP と同様に (n/g) 個の頂点だけを処理の対象にすることにより，計算量の削減を行うことができる．対称性を持つグラフを用いた場合の BFS-APSP の計算量は $O(n^2d/g)$ であり，ADJ-APSP の計算量は，$O(n^2dD/gE)$ である．ただし，処理の対象になる頂点数が減るため，MPI の最大並列数も減ることに注意が必要である．MPI の最大並列数は BFS-APSP は (n/g) であり，ADJ-APSP は $\lceil(n/g)/E\rceil$ で

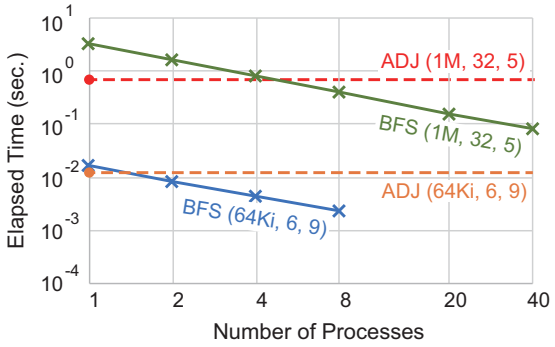


(a) On the K computer

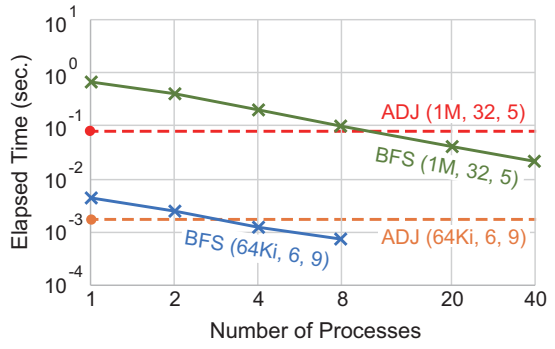


(b) On the Cygnus system

図 17: Results for Symmetrical Graphs on Single Node



(a) On the K computer



(b) On the Cygnus system

図 18: Results for Symmetrical Graphs on Multi Nodes

ある．OpenMP の最大並列数は，3 章で説明した各 APSP アルゴリズムと同じ n である．

次に，対称性を持つグラフを用いた各 APSP アルゴリズムの性能評価を行う．評価に用いるグラフは，本節の第一段落の手順によって作成した対称性を持つグラフ (n, d, D) = (64Ki, 6, 9) と (1M, 32, 5) である．それらのグラフに対してメタヒューリスティクスを用いて最適化を行ったため，直径 D の値は前節までに利用したランダムグラフと同じ値になった．実験では，4.3.2 節と同様にスレッド数は各計算機環境の最大値に設定する．そしてプロセス数を 1 に固定し，グループ数 g のみを変化させた．なお，g の最大値は，(64Ki, 6, 9) は 8,192 であり，(1M, 32, 5) は 25,000 である．

評価結果を**図 17** に示す．この結果より，すべての場合において，ADJ-APSP の方が BFS-APSP よりも高速であることがわかる．しかし，グループ数 g の値が大きい場合に ADJ-APSP の速度向上率は BFS-APSP と比較して低くなることがわかる．その理由は 4.3.2 節と同じであり，グループ数 g が増えるにつれ，隣接行列の列の要素数が少なくなるからである．また，(64Ki, 6, 9) においては $g = 1,024$ 以降の計測時間は変わらないことがわかる．その理由は，$g = 1,024$ の場合の (64Ki, 6, 9) で用いる隣接行列の列の要素数は 1 だからであり，$g = 1024$ 以降は (64Ki, 6, 9) における ADJ-APSP の計算量は変化しない．

次に，スレッド数とグループ数 g の値を最大値に固定し，MPI のプロセス数を変化させた結果を**図 18** に示す．(64Ki, 6, 9) と (1M, 32, 5) の MPI の最大並列数は，BFS-APSP は 8 と 40 であり，ADJ-APSP は共に 1 である．そのため，図 18 では，ADJ-APSP の 1 プロセス時の値を点線で表示している．この結果より，すべての場合において，BFS-APSP の方が ADJ-APSP よりも高速であることがわかる．(64Ki, 6, 9) と (1M, 32, 5) の BFS-APSP における最良値は，「京」は 2.41×10^{-3} 秒と 7.90×10^{-2} 秒であり，Cygnus は 7.35×10^{-4} 秒と 2.20×10^{-2} 秒である．

6. まとめと今後の課題

本稿では，APSP アルゴリズムである BFS-APSP と ADJ-APSP に着目し，それぞれについて MPI と OpenMP を用いた並列化を行った．性能比較の結果，逐次アルゴリズムおよび OpenMP を用いたスレッド並列アルゴリズムにおいては，ADJ-APSP の方が性能が高いことを示した．しかしながら，MPI の最大並列数は BFS-APSP の方が ADJ-APSP よりも多いため，MPI を用いた並列アルゴリズムにおいては，BFS-APSP の方が性能が高くなる場合があることを示した．さらに，ADJ-APSP について GPU を用いた並列化を行うことにより，CPU よりも最大 16.53 倍の性能向上を達成した．また，対称性を持つグラフに対して各 APSP アルゴリズムを適用させることで，計算量を大幅に削減できることを示した．対称性を持つグラフを用いる場合は，特に ADJ-APSP の MPI の最大並列数が少な

くなるため，比較的小さい MPI の並列数で BFS-APSP の性能は ADJ-APSP を上回ることを示した．

今後の課題として，下記が挙げられる．(1) 各 APSP アルゴリズムに対するさらなる並列化を行う．本稿の BFS-APSP の並列化において，単体の BFS についてはスレッド並列化しか行っていないが，単体の BFS についても MPI による並列化が可能である [21]．また，ADJ-APSP についても，隣接行列を横に分割することにより，さらなる並列化が可能である．(2) ユーザに対する利便性の向上のため，入力グラフのサイズや計算機環境などに応じて高速な APSP アルゴリズムを自動的に選択する機能を作成する．この機能の作成のためには，各 APSP アルゴリズムのモデル化を行う必要があると考えられる．

Acknowledgements

This research used the Cygnus system provided by Interdisciplinary Computational Science Program in the Center for Computational Sciences, University of Tsukuba. This work was supported by JSPS KAKENHI Grant Number 18K11331 and RIKEN Incentive Research Projects. We express our sincere thanks to Dr. Ryuhei Mori for an excellent implementation of serial version ADJ-APSP. We are grateful to the Graph Golf committee for giving us an interesting issue.

参考文献

[1] Takafumi Watanabe and Masahiro Nakao and Tomoyuki Hiroyasu and Tomohiro Otsuka and Michihiro Koibuchi. Impact of topology and link aggregation on a pc cluster with ethernet. In *2008 IEEE International Conference on Cluster Computing*, pp. 280–285, Sep. 2008.

[2] Ankit Singla and Chi-Yao Hong and Lucian Popa and Philip Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. *CoRR*, Vol. abs/1110.1687, , 2011.

[3] Shin, Ji-Yong and Wong, Bernard and Sirer, Emin Gün. Small-world Datacenters. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pp. 2:1–2:13, New York, NY, USA, 2011. ACM.

[4] M. Koibuchi and H. Matsutani and H. Amano and D. F. Hsu and H. Casanova. A case for random shortcut topologies for HPC interconnects. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 177–188, June 2012.

[5] Michihiro Koibuchi and Ikki Fujiwara and Kiyo Ishii and Shu Namiki and Fabien Chaix and Hiroki Matsutani and Hideharu Amano and Tomohiro Kudoh. Optical network technologies for HPC: computer-architects point of view. *IEICE Electronics Express*, Vol. 13, No. 6, pp. 1–14, 2016.

[6] H. Matsutani and M. Koibuchi and I. Fujiwara and T. Kagami and Y. Take and T. Kuroda and P. Bogdan and R. Marculescu and H. Amano. Low-latency wireless 3D NoCs via randomized shortcut chips. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[7] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, Vol. 21, No. 6, pp. 1087–1092, 1953.

[8] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, Vol. 220, No. 4598, pp. 671–680, 1983.

[9] Nakao, Masahiro and Murai, Hitoshi and Sato, Mitsuhisa. A method for order/degree problem based on graph symmetry and simulated annealing with mpi/openmp parallelization. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2019, pp. 128–137. ACM, 2019.

[10] Michihiro Koibuchi and Ikki Fujiwara and Shoichi Hirasawa and Satoshi Fujita and Koji Nakano and Takeaki Uno and Takeru Inoue and Ken-ichi Kawarabayashi. Graph Golf: The Order/degree Problem Competition, 2018. `http://research.nii.ac.jp/graphgolf`.

[11] Nobutaka Shimizu and Ryuhei Mori. Average Shortest Path Length of Graphs of Diameter 3. *CoRR*, Vol. abs/1606.05119, , 2016.

[12] Teruaki Kitasuka and Masahiro Iida. A Heuristic Method of Generating Diameter 3 Graphs for Order/Degree Problem. *CoRR*, Vol. abs/1609.03136, , 2016.

[13] R. Mizuno and Y. Ishida. Constructing large-scale low-latency network from small optimal networks. In *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pp. 1–5, Aug 2016.

[14] Warshall, Stephen. A Theorem on Boolean Matrices. *J. ACM*, Vol. 9, No. 1, pp. 11–12, January 1962.

[15] Floyd, Robert W. Algorithm 97: Shortest Path. *Commun. ACM*, Vol. 5, No. 6, pp. 345–, June 1962.

[16] R. Seidel. On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs. *Journal of Computer and System Sciences*, Vol. 51, No. 3, pp. 400–403, 1995.

[17] 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, pp. 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[18] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 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, pp. 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

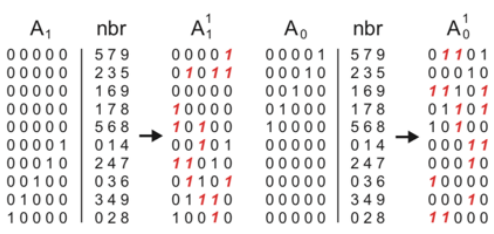
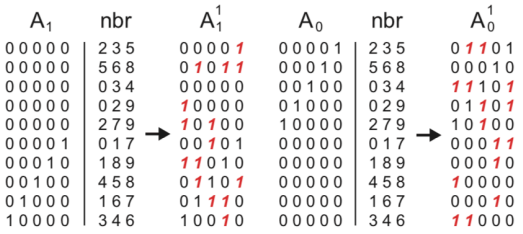
[19] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *Knowledge Discovery in Databases: PKDD 2005*, pp. 133–145, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[20] Ryuhei Mori. `https://github.com/ryuhei-mori/graph_ASPL`.

[21] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pp. 25–25, Nov 2005.

正誤表

下記の箇所に誤りがございました．お詫びして訂正いたします．

訂正箇所	誤	正
3ページ左25行目	なお，MPI および OpenMP の最大並列数は，共に n である．	なお，MPI の最大並列数は n である．
4ページ図6	 図 6: Example of Divided ADJ-APSP	 図 6: Example of Divided ADJ-APSP
5ページ左21行目	なお，MPI の最大並列数は隣接行列の列の要素数と同じ [n/E] であり，OpenMP の最大並列数は n である．	なお，MPI の最大並列数は隣接行列の列の要素数と同じ [n/E] ある．
9ページ左1行目	OpenMP の最大並列数は，3 章で説明した各 APSP アルゴリズムと同じ n である．	（削除）