Vol.2020-HPC-175 No.19
2020/7/31

スーパーコンピュータ「富岳」における Graph500 ベンチマークの幅優先探索の性能評価

中尾 昌広[1,a)]　上野 晃司[2]　藤澤 克樹[3]　児玉 祐悦[1]　佐藤 三久[1]

概要：様々な分野で大規模グラフを高速に処理する需要が高まっている．しかしながら，グラフ処理は一般的に不規則な計算が必要であるため，大規模分散メモリシステムにおいてグラフ処理の性能をスケールさせることは難しい．そのような背景から，大規模グラフ処理の性能を評価するランキングである Graph500 が誕生した．我々は，Graph500 ベンチマークで採用されているカーネルの 1 つである幅優先探索（BFS : Breadth-First Search）に対する改良を行ってきており，「京」を用いた結果で 9 期連続（通算 10 期）の世界 1 位を獲得してきた．そこで，本稿では，2021 年に共用開始予定の「富岳」を用いた BFS の性能チューニングおよび評価を行った．約 1.1 兆個の頂点と 17.6 兆本のエッジから構成される大規模グラフに対する BFS を「富岳」の 92,160 ノードを用いて評価を行った結果，「京」の 2.27 倍の性能を達成し，2020 年 6 月の Graph500 で 1 位になった．また，不規則な計算が大半を占める BFS においても「富岳」は高い性能を発揮できたため，この結果は「富岳」の汎用性の高さを示しているとも言える．

1. はじめに

実社会における大規模データをグラフ（要素間の関係性を頂点とエッジで表したデータ構造）に変換し，それを計算機で高速処理する需要が高まっている．例えば，SNS におけるユーザ間の繋がりの分析，検索エンジンの高性能化，サイバーセキュリティ，VLSI のレイアウトや道路網の最適化，創薬，遺伝子解析，全脳シミュレーションなど様々な分野でグラフ処理が用いられている [1–3]．また，IoT（Internet of Things）の発展により，大量のセンサから送信される実データを解析するためにもグラフ処理が用いられる．それらのグラフの頂点数は 1 兆点を超えることがあり，またそのエッジの数は頂点数の数十倍になることもある [4]．

このような背景から，大規模グラフ処理の性能を評価するランキングである Graph500 が誕生した [5, 6]．Graph500 では．Kronecker グラフ [7] というスケールフリー性のあるグラフが用いられる．スケールフリー性とは，一部の頂点が他の多くの頂点と繋がっている一方，他の多くの頂点はごく僅かな頂点としか繋がっていない性質のことであり，SNS における実データも同じような性質を持つことが知られている．Graph500 は 2010 年に誕生し，年に 2 回（6 月と 11 月）更新されている．Graph500 で用いられるベンチマークは (1) グラフ構築，(2) 幅優先探索（BFS : Breadth-First Search），(3) 単一始点最短経路（SSSP : Single-Source Shortest Path）の 3 つのカーネルから構成されている．なお，SSSP は 2017 年 11 月に追加された．グラフ構築はランキングに関係なく，BFS と SSSP についてのみ個別のランキングが行われる．本稿では BFS について述べる．BFS はグラフの強連結成分分解や中心性解析などで用いられる重要なアルゴリズムである．

我々が開発した BFS の評価をスーパーコンピュータ「京」[8] で行った結果，2014 年 6 月と 2015 年 6 月〜2019 年 6 月の Graph500 において，9 期連続（通算 10 期）の世界 1 位を獲得していた．「京」の運用終了に伴い，2020 年 11 月以降は「京」のランキングは Graph500 から外れている．そして，2020 年 6 月，「京」の後継機であるスーパーコンピュータ「富岳」[9] の一部を用いた結果により，我々は再び世界 1 位のランキングを獲得した．本稿では，Graph500 の投稿に用いた BFS と，「富岳」における BFS の性能チューニングについて述べる．

本稿の構成は下記の通りである．2 章では，Graph500 でよく利用されている Hybrid BFS について述べる．3 章では，Hybrid BFS をベースに我々が開発した BFS を紹介する．4 章では「富岳」について，5 章では「富岳」における BFS の性能チューニングについて述べる．6 章では，まとめと今後の課題について述べる．

[1] RIKEN Center for Computational Science
[2] Fixstars Corporation
[3] Institute of Mathematics for Industry, Kyushu University
[a)] masahiro.nakao@riken.jp

```
1  hybrid-bfs(vertices, source, neighbors)
2   frontier ← {source}
3   next ← {}
4   parents ← [-1,-1,...,-1]
5   while frontier ≠ {} do
6   | if next-direction(...) = top-down then
7   | | top-down(vertices, frontier, next, parents, neighbors)
8   | else
9   | | bottom-up(vertices, frontier, next, parents, neighbors)
10  | frontier ← next
11  | next ← {}
12  return parents
13
14 top-down(vertices, frontier, next, parents, neighbors)
15  for v ∈ frontier do
16  | for n ∈ neighbors[v] do
17  | | if parents[n] = -1 then
18  | | | parents[n] ← v
19  | | | next ← next ∪ {n}
20
21 bottom-up(vertices, frontier, next, parents, neighbors)
22  for v ∈ vertices do
23  | if parents[v] = -1 then
24  | | for n ∈ neighbors[v] do
25  | | | if n ∈ frontier then
26  | | | | parents[v] ← n
27  | | | | next ← next ∪ {v}
28  | | | | break
```

図 1: Hybrid BFS [10]

2. Hybrid BFS

2.1 共有メモリシステム用のアルゴリズム

Hybrid BFS の擬似コードを**図 1** に示す [10]．2 行目では，1 つの出発点（*source*）を訪問点集合（*frontier*）に代入している．3 行目では，次訪問点集合（*next*）を空集合として初期化している．4 行目では，最終出力である BFS ツリー（*parents*）を初期化している．なお，*parents* に代入している “-1” は未訪問であることを意味する．

従来の BFS は，関数 **top-down()** だけで構成されている．関数 **top-down()** で行われる操作を Top-down アプローチと呼ぶ．Top-down アプローチでは，まず *frontier* と隣接する頂点が訪問済であるかをチェックする（15〜17 行目）．なお，*neighbors* は隣接頂点集合である．もし未訪問であれば，その未訪問頂点の接続元を *parents* に代入する（18 行目）．また，その未訪問頂点を *next* に重複なく追加する（19 行目）．

Kronecker グラフにおける Top-down アプローチは，探索序盤と終盤は高速であるが，探索中盤の効率が悪いことが知られている．その原因は，*frontier* と隣接する全頂点に対して訪問済かどうかのチェックを行うとき（15〜17 行目），探索中盤では *frontier* の要素数が多く，さらに *frontier* の隣接頂点はすでに訪問済もしくは *frontier* に属している場合が多いため，冗長なチェックが増えてしまうことが原因である．

$$\begin{matrix} A_{1,1} & A_{1,2} & \cdots & A_{1,C} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ A_{R,1} & A_{R,2} & \cdots & A_{R,C} \end{matrix}$$

図 2: Distribution of adjacency matrix

その問題点を克服するため，Hybrid BFS では，従来の Top-down アプローチと新しい Bottom-up アプローチを切り替えながら BFS を実行する．Bottom-up アプローチは関数 **bottom-up()** で行われる．Bottom-up アプローチは，Top-down アプローチとは逆方向に探索を行う．具体的には，Top-down アプローチでは *frontier* を始点として，その隣接する未訪問頂点を探すのに対し，Bottom-up アプローチでは全ての未訪問頂点を始点として，その隣接する頂点の中に *frontier* に属する頂点を探す（22〜25 行目）．*frontier* に属する頂点が見つかった場合は，その頂点を *parents* に代入し，その頂点の始点を *next* に重複なく追加する（26〜27 行目）．

Bottom-up アプローチの利点は，*frontier* に属する頂点が 1 つでも見つかった時点で，その始点に対する探索を終了できるため（28 行目），Top-down アプローチに見られる冗長なチェックを削減できる点である．ただし，Bottom-up アプローチは全頂点が未訪問であるかをチェックする必要があるため，*frontier* が小さいときは，Top-down アプローチの方が高速である．そのため，Hybrid BFS は *frontier* が小さい場合は Top-down アプローチを用い，*frontier* が大きい場合は Bottom-up アプローチを用いる．詳細は省略しているが，6 行目の関数 **next-direction()** は，*frontier* に属する頂点数などの情報を用いて，Top-down アプローチと Bottom-up アプローチの切り替えを動的に決定する．

実装では，1 頂点を 1 ビットで表したビットマップやスレッド並列を用いた高速化がよく行われる [11]．なお，スレッド並列化において，Top-down アプローチでは *parents* の更新にアトミック操作が必要なのに対し，Bottom-up アプローチではアトミック操作は必要ないため，並列化効率は Bottom-up アプローチの方が高い．

2.2 分散メモリシステム用のアルゴリズム

大規模グラフを扱うため，分散メモリシステムで動作する並列版 Hybrid BFS が提案されている [12]．並列版 Hybrid BFS では，**図 2** に示すように，2 次元に分割されたプロセス（R 行 C 列）に隣接行列 A を割り当てる．あるプロセス $P(i,j)$ は部分的な隣接行列 $A_{i,j}$ の情報を持つ．Graph500 では無向グラフを扱うため，隣接行列 A は対称行列である．隣接行列 A の k 列 l 行目に非ゼロ要素がある場合，k 番と l 番の頂点が隣接していることを意味する．

```
1  parallel-top-down(...)
2   f ← {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   | | t_{i,j} ← {}
10  | | for u ∈ f_i do
11  | | | for v ∈ A_{i,j}(:, u) do
12  | | | | t_{i,j} ← t_{i,j} ∪ (u, v)
13  | | t_{i,j} ← alltoallv(t_{i,j}, P(i, :))
14  | | for (u, v) ∈ t_{i,j} do
15  | | | if π_{i,j}(v) = −1 then
16  | | | | π_{i,j}(v) ← u
17  | | | | n_{i,j} ← n_{i.j} ∪ v
18  | | f ← n
19  | | n ←{}
20  return π
```

図 3: Parallel Top-down approach [12]

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

図 4: Parallel Bottom-up approach [12]

並列版の Top-down アプローチと Bottom-up アプローチの擬似コードを**図 3** と**図 4** に示す．並列化版 Hybrid BFS は図 1 に示した逐次版 Hybrid BFS と同様に，図 3 と図 4 の手法を切り替えて実行する．f，n，π は，図 1 の $frontier$，$next$，$parents$ にそれぞれ対応する．t は隣接する 2 つの頂点（u と v）を一時的に保持するための疎ベクトルである．c はチェック済の頂点を表すビットマップである．f と n は，Top-down アプローチでは疎ベクトルが用いられ，Bottom-up アプローチではビットマップが用いられる．π は両アプローチで密ベクトルが用いられる．

表 1: Compressed Sparse Row [13]

row-starts	0	2	2	2	2	2	2	3	4
dst	4	5	3	1					

表 2: Bitmap-based Compressed Sparse Row [13]

row-starts	0	2	3	4				
bitmap	1	0	0	0	0	0	1	1
$offset$	0	1	3					
dst	4	5	3	1				

図 3 に示す Top-down アプローチの 7〜8 行目では，f を列プロセス内で共有している．9〜13 行目では，f と隣接する頂点の情報を行プロセス内で交換している．14〜17 行目では，π と n を作成している．図 4 に示す Bottom-up アプローチの 8〜9 行目は，f のデータ構造を除いて図 3 の 7〜8 行目と同じである．10 行目以降の処理は，C 個（列数）のサブステップに分割して行う．サブステップに分割する理由は，22 行目で c の更新を行プロセス内で定期的に行うことにより，各プロセスで探索の対象となる頂点を減らし，全体的な高速化を行うためである．10〜18 行目では，f と隣接している未訪問頂点の情報を行プロセス内で交換している．19〜21 行目では，π と n を作成している．

3. Hybrid BFS の改良

本章では，2.2 節で述べた並列 Hybrid BFS に改良を加えた BFS アルゴリズムの概要を紹介する．その詳細については文献 [13] を参考にされたい．

3.1 隣接行列におけるビットマップを使った疎行列表現

一般的な CSR（Compressed Sparse Row）を隣接行列の格納形式として用いる場合，出力先の頂点番号を保持する配列 *dst* とエッジの頂点番号のオフセット配列 *row-starts* を用いる．エッジの情報を効率よく取り出すためには，*row-starts* のサイズは小さい方が望ましいが，R 行 C 列の 2 次元分割を行う場合の *row-starts* のサイズは C に比例して大きくなるという問題点がある．

そこで，CSR よりも省メモリかつ効率良くエッジの情報を取り出すことが可能なビットマップを使った疎行列表現 BCSR（Bitmap-based CSR）を開発した．BCSR では，次の 3 つの工夫を行っている．(1) 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する．(2) 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 ビットで表した配列 *bitmap* を用いる．(3) エッジの入力元の頂点番号を効率良く計算するための配列 $offset$ を用いる．ある頂点 v の *row-starts* の位置は，*bitmap* の先頭から頂点 v に対応するビットまでの立っているビット数である．*bitmap* の立っているビットを効率的に計算するため，あらかじめワード単位で立っているビット数の累計を $offset$ に保存する．

表 3: Memory consumption in CSR and BCSR [13]

	CSR		BCSR	
	Order	Actual	Order	Actual
row-starts	-	-	$n'C/64$	32MB
bitmap	-	-	$n'C/64$	32MB
$offset$	$n'C$	2048MB	$n'p$	190MB
dst	$n'd$	1020MB	$n'd$	1020MB
TOTAL	$n'(C+d)$	3068MB	$n'(\frac{C}{32}+p+d)$	1274MB

エッジリストが {(0, 4), (0, 5), (6, 3), (7, 1)} の場合の CSR と BCSR の例を**表 1** と**表 2** に示す．表 2 では，説明のため 1 ワードは 4 ビットとしている．CSR の *row-starts* が，BCSR では *row-starts*，*bitmap*，$offset$ の 3 つの配列で表現されている．次にメモリ量の比較を**表 3** に示す．表 3 では，1 ワードは 64 ビットとしている．表中の n' は 1 プロセスが持つ頂点数，d は次数，p は 1 プロセスが持つ部分隣接行列から 1 行取り出したときにエッジが 1 本以上存在する確率である．$R \times C = 64 \times 32$ の 2 次元分割を行った場合の 160 億頂点 2560 億エッジの Kronecker グラフを用いた実際のメモリ使用量も示す．この結果より，BCSR は CSR よりも省メモリであることがわかる．

3.2 頂点番号の並べ替え

ビットマップのビットの位置は一般的には頂点番号順である．そして，Kronecker グラフは次数の大きい頂点と小さい頂点を持ち，BFS では次数の大きい頂点ほど頻繁にアクセスされる．そこで，ビットマップのビットの位置を頂点の次数順にすることにより，メモリの局所性による高速化を図ることができる．実装としては，各プロセスが持つ部分隣接行列に対して頂点の次数を計算し，プロセス内でのみ次数順に頂点番号の再割り当てを行う．ただし，この方法だと計算結果である *parents* も次数順の番号で作成されてしまう．そこで，元の頂点番号を保持した配列を用意しておき，*parents* に書き込む時は元の頂点番号を用いる．この手法の副次的な効果として，エッジを持たない頂点を計算から除外して，計算に用いるデータのサイズを小さくすることが可能である．

3.3 隣接行列における分割方法の改良

図 2 に示した 2 次元ブロック分割を隣接行列に適用すると，図 3 の 7 行目および図 4 の 8 行目において，*frontier* を転置するための **transpose-vector()** による通信が必要になる．そこで，Yoo らが提案している分割方法 [14] を隣接行列に適用することで，それらの **transpose-vector()** を省くことができる．その分割方法を**図 5** に示す．隣接行列の行に対する分割は図 2 と同様に C 個のブロック分割であるのに対し，隣接行列の列に対する分割は $R \times C$ 個のブロックサイクリック分割になるように行う．すなわち，

$$\begin{matrix} A^{(1)}_{1,1} & A^{(1)}_{1,2} & \cdots & A^{(1)}_{1,C} \\ A^{(1)}_{2,1} & A^{(1)}_{2,2} & \cdots & A^{(1)}_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(1)}_{R,1} & A^{(1)}_{R,2} & \cdots & A^{(1)}_{R,C} \\ A^{(2)}_{1,1} & A^{(2)}_{1,2} & \cdots & A^{(2)}_{1,C} \\ \vdots & \vdots & \ddots & \vdots \\ A^{(C)}_{R,1} & A^{(C)}_{R,2} & \cdots & A^{(C)}_{R,C} \end{matrix}$$

図 5: Yoo's distribution of adjacency matrix [14]

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

図 6: Simple thread parallelization [13]

```
1 top-down-sender-load-balanced(f_i, A_{i,j})
2  for u ∈ f_i in parallel do
3  | for k ∈ P(i, :) do
4  | | (v_0, v_1) ← edge-range(A_{i,j}(:, u), k)
5  | | r_{i,j,k} ← r_{i,j,k} ∪ (u, v_0, v_1)
6  for k ∈ P(i, :) in parallel do
7  | for (u, v_0, v_1) ∈ r_{i,j,k} do
8  | | for v ∈ A_{i,j}(v_0:v_1, u) do
9  | | | t_{i,j,k} ← t_{i,j,k} ∪ (u, v)
```

図 7: Proposed thread parallelization [13]

各プロセスは C 個の分割された隣接行列を持つ．この分割方法の実装は図 2 と比べて複雑であるが，計算オーバヘッドはないという特徴を持つ．

3.4 Top-Down アプローチのロードバランス最適化

図 3 の 10〜12 行目のスレッド実装の単純な例を**図 6** に示す．2 行目では，*frontier* に含まれる入力元頂点でスレッド分割している．3 行目の $A_{i,j}(:, u)$ は，エッジの入力元が u であるエッジリストである．4 行目の関数 **owner**(v) は，出力先頂点 v の担当プロセスを返す関数である．5 行目では，隣接頂点情報を保存している．図 3 の手法は簡易であるが，Kronecker グラフでは頂点によって次数は大きく異なるため，スレッド間で大きなロードインバランスが発生するという問題点がある．

そこで，ロードインバランスを小さくするため，出力先頂点でスレッド分割する実装を**図 7** に示す．この手法はスレッド並列された 2 つの for ループ文を用いる．最初の for ループ文で担当プロセス毎の出力先頂点の情報を保存し，2 つ目の for ループ文で隣接頂点の組を保存する．4 行目の関数 **edge-range**($A_{i,j}(:, u)$, k) は担当プロセスが k であるエッジリストの範囲を返す関数である．

```
1  parallel-bottom-up(...)
2   f ← {source}
3   c ← {source}
4   n ← {}
5   π ← [-1,-1,...,-1]
6   for all compute nodes P(i, j) in parallel do
7   | while f ≠ {} do
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} ∪ (u, v)
15  | | | | | | c_{i,j} ← 1
16  | | | | | | break
17  | | | c_{i,j} ← sendrecv(c_{i,j}, P(i, j+1), P(i, j-1))
18  | | t_{i,j} ← alltoallv(t_{i,j}, P(i, :))
19  | | for (v, u) ∈ t_{i,j} do
20  | | | π_{i,j}(v) ← u
21  | | | n_{i,j} ← n_{i.j} ∪ v
22  | | f ← n
23  | | n ←{}
24  return π
```

図 8: Proposed Parallel Bottom-up approach [12]

図 6 の手法では，一時バッファに隣接頂点の組を入れて，**alltoallv** 通信用のメモリにコピーするという実装になっている．通信用メモリに直接コピーできない理由は，**alltoallv** 通信用のメモリは連続している必要があるが，各プロセスに送信するための要素数が事前にわからないからである．それに対し，図 7 の手法では，最初の for ループ文において各プロセスに渡す頂点数も計上できるので，9 行目では一時バッファを介さずに隣接頂点の組を通信用メモリに直接保存できるというメリットがある．しかしながら，図 7 の手法のデメリットとして，t よりも r の方が情報量が大きいため，送信先プロセス数と比べて次数が比較的小さい頂点を探索する場合，t よりも r に書き込む量の方が大きくなり，効率が悪くなる点が挙げられる．そこで，実装では，頂点毎に図 6 と図 7 の手法を選択している．

3.5 Bottom-up アプローチの通信最適化

3.5.1 集合通信の利用

図 4 の 18 行目の **sendrecv** 通信では，プロセス行内で 1 対 1 通信を行っている．大規模環境における予備実験の結果，このようなスケジューリングされていない 1 対 1 通信が多発すると，通信の効率が悪くなることがわかった．そこで，図 4 の関数 **parallel-bottom-up()** を**図 8** のように変更した．図 4 の 18 行目の **sendrecv** 通信の代わりに図 8 の 18 行目の **alltoallv** 通信を使うことで，まとめてデータ交換するようにしている．また，3.3 節で述べた通り，図 8 では図 4 の 8 行目の関数 **transpose-vector()** を省いている．

表 4: Top-down communication costs [12]

	Times/Step	Words/Search
allgatherv	$O(1)$	nR
alltoallv	$O(1)$	$4m$

表 5: Bottom-up communication costs [12]

	Times/Step	Words/Search
allgatherv	$O(1)$	$s_b nR/64$
sendrecv	$O(C)$	$s_b nC/64$
alltoallv	$O(1)$	$2n$

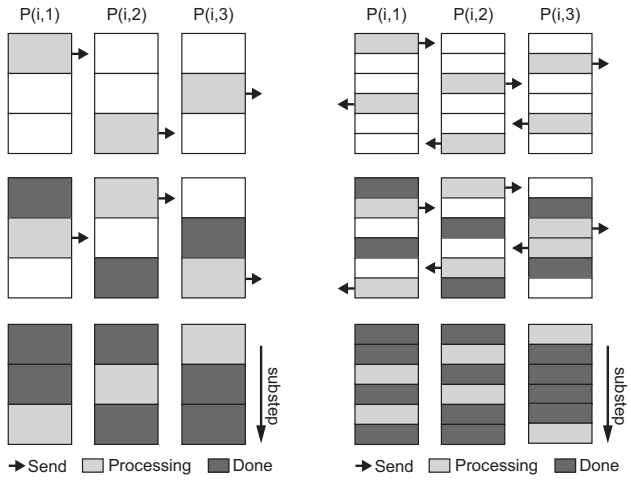
3.5.2 頂点濃度によるデータ構造の切り替え

3.3 節で述べた隣接行列の分散を行った場合の各アプローチの通信コストを**表 4** と**表 5** に示す．各表の 2 列目は，while 文の 1 ステップを計算するのに必要な通信回数を示している．各表の 3 列目は，アプローチの切り替えを行わないと仮定した場合の 1 回の BFS を行うのに必要な通信サイズを示している．ここで，1 ワードは 64 ビット，n は頂点数，R と C はプロセスの分割数，m はエッジ数，s_b は while 文のステップ数である．また，式の簡易化のため，$(C-1)/C \approx 1$ としており，1 ワードだけの通信は除いている．

表 5 の **allgatherv** と **sendrecv** は，ビットマップを用いているため，1 ステップにおける通信サイズが R や C に比例して増えるという問題点がある．なお，表 4 の **allgatherv** は疎ベクトルを用いており，$frontier$ が小さい場合のみ発行されるので問題にはならない．そこで，表 5 の **allgatherv** と **sendrecv** の通信サイズを削減するため，データの頂点濃度に応じてビットマップもしくは疎ベクトルを自動的に選択する実装を行う．各通信に疎ベクトルを用いた場合，**allgatherv** の通信サイズは $frontier$ の頂点数に比例し，**sendrecv** の通信サイズは未訪問の頂点数に比例する．すなわち，それぞれの頂点数が $n/64$ より小さい場合に疎ベクトルを用いることで，それぞれの通信サイズを削減できる．

3.5.3 通信と計算のオーバラップ

図 8 の 17 行目の **sendrecv** において通信と計算のオーバラップを行うため，図 8 の 9 行目のサブステップ数を C から n 倍の nC に増やし，**sendrecv** を n 個同時に実行可能にする．実装では $n = 4$ を用いている．また，「富岳」や BlueGene/Q [15] などの大規模分散メモリシステムで用いられているトーラス形状の直接網を有効利用するため，**sendrecv** 通信を 2 方向同時に行う．$C = 3$ の場合の通信と計算の概念図を**図 9** に示す．**図 9a** は従来の $n = 1$，**図 9b** は新規の $n = 2$ の場合である．なお，通信待ち時間を減らすため，受信プロセスが処理する順番はループの順番とは関係なく，受信した順に処理する．



(a) $n = 1$ (b) $n = 2$

図 9: Overlapping computation with communication

表 6: Specification of compute node of Fugaku [9]

CPU	A64FX, 48+2/4 cores, 2.0/2.2GHz, 3,072/3,379GFlops(DP)
Memory	HBM2 32 GiB, 1,024GB/s
Interconnect	Tofu-D, 6-dimensional mesh/torus 28.05Gbps × 2 lane ×10 ports

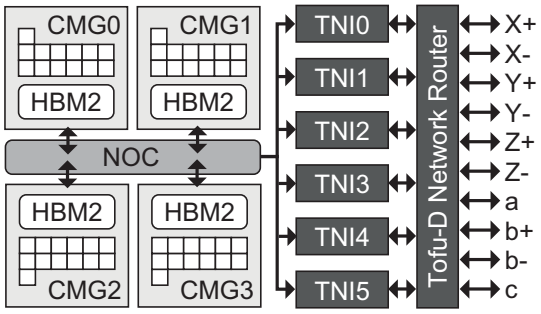


図 10: Block diagram of A64FX processor [17]

4. スーパーコンピュータ「富岳」

「京」の後継機である「富岳」は理化学研究所 計算科学研究センター [16] に設置されているスーパーコンピュータであり，2021 年度に共用開始予定である．「富岳」のスペックを**表 6** に示す．

「富岳」の CPU である A64FX [18] は Armv8.2-A アーキテクチャをベースとし，SVE（Scalable Vector Extension）や HBM2（High Bandwidth Memory ver.2）を利用することで，高い演算性能とメモリバンド幅を実現した汎用プロセッサである．A64FX の構成を**図 10** に示す．「富岳」には計算処理を行う “計算ノード” と，計算処理と I/O 処理を行う “計算ノード兼 I/O ノード” がある．各ノードの A64FX は 48 個の計算コアを持ち，さらに計算ノードと計算ノード兼 I/O ノードは 2 個と 4 個のアシスタントコアをそれぞれ利用する．アシスタントコアは OS や通信などの割り込みが発生する処理を行う．コアの動作クロック数はユーザがジョブ毎に 2.0GHz もしくは 2.2GHz を選択できる．それぞれの倍精度のピーク性能は 3,072GFlops と 3,379GFlops である．A64FX では，12+1 個のコアと 8GiB の HBM2 で構成される CMG（Core Memory Group）が 4 つで構成されている．4 つの CMG は NOC（Network on Chip）で接続されているため，A64FX は 4NUMA ノード構成を取る．このことから，一般的なアプリケーションに対するノードあたりのプロセス数は 4 の約数もしくは 4 の倍数が推奨されている．

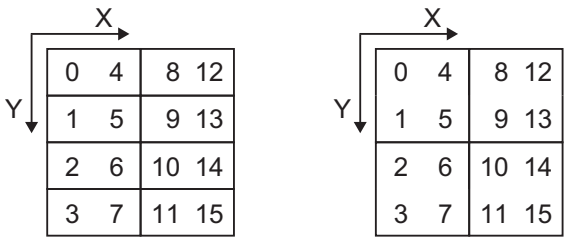
「富岳」のインターコネクトには Tofu Interconnect D（Tofu-D）[17] が用いられている．Tofu-D の形状は 6 次元メッシュ/トーラスであり，$XYZabc$ の 6 次元によりノード位置を特定できる．a と c は 2 ノードのみで構成されるメッシュトポロジであり，それ以外はトーラストポロジである．6 次元の内の abc の大きさは $(a, b, c) = (2, 3, 2)$ で固定あり，XYZ の大きさはシステム依存である．なお，「富岳」は $(X, Y, Z) = (24, 23, 24)$ であるため，総ノード数は 158,976 である．また，図 10 の通り，A64FX は Tofu-D 用の 10 個のポートを持ち，$XYZb$ 軸はそれぞれ 2 つのポートを使用し，ac 軸はそれぞれ 1 つのポートを使用する．A64FX は 6 つの TNI（Tofu Network Interface）を持ち，6 方向に同時に 6.8GB/s の通信が可能である．すなわち，「富岳」の各ノードのインジェクションバンド幅は 40.8GB/s である．

比較として，「京」のノードの TNI の数は 4 であり，各リンクの通信速度は 5.0GB/s であるため，インジェクションバンド幅は 20.0GB/s であった．また，レイテンシ（8B Put 通信）については，「富岳」は 0.49〜0.54μs に対し，「京」は 0.91〜1.15μs である [17].

5. 性能評価

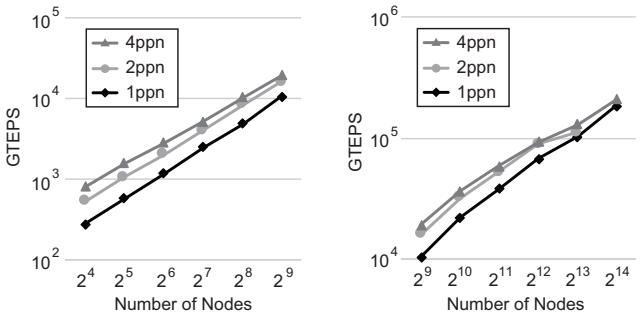
5.1 Graph500 ベンチマーク

Graph500 で用いられるグラフの頂点数は 2 のべき乗であり，2^{SCALE} と表現する．$SCALE$ を問題サイズと呼ぶ．エッジ数は頂点数の 16 倍である．BFS の性能単位は TEPS（Traversed Edges Per Second）であり，1 秒間に探索したエッジ数を表す．例えば，1GTEPS は，1 秒間に 10 億エッジを探索できることを指す．BFS では，探索開始点としてランダムに 64 個の頂点が選ばれ，それぞれ順番に BFS が行われる．そして，64 回の BFS における各性能の調和平均が性能値に用いられる．本章で行う性能チューニングでは 64 回は多いため，5.6 節で行う最終評価を除いて 16 回の BFS における調和平均を性能値として用いる．我々が開発したコードは `https://github.com/suzumura/graph500` で公開している．



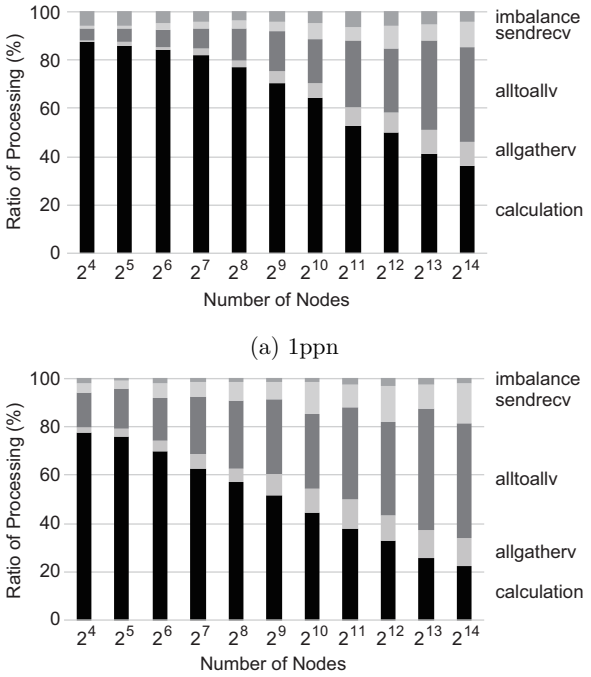
(a) 2ppn $(Y, X) = (4, 2)$ (b) 4ppn $(Y, X) = (2, 2)$

図 11: Example of process mapping $(R, C) = (4, 4)$



(a) 16 to 512 nodes (b) 512 to 16,384 nodes

図 12: Performance for each process per node



(a) 1ppn

(b) 4ppn

図 13: Time ratio of processing

5.2 ジョブの設定

ノードあたりのグラフの問題サイズは $SCALE = 24$ とし，弱スケーリングで計測する．「富岳」のジョブスクリプトでは，1〜3 次元の論理的なプロセス配置（ジョブ形状）を指定できる．BFS では $R \times C$ の 2 次元プロセスグリッドを用いるので，2 次元のジョブ形状を指定する．この場合，各プロセスは物理的に 2 次元トーラス形状になるようにノードに割り当てられる．なお，表 4 と表 5 より，R と C の値が近い場合に通信サイズは小さくなることがわかる．しかしながら，列プロセス内で発生する **allgatherv** よりも行プロセス内で発生する **alltoallv** の方が通信コストは大きいため，$R = C$ にできない場合は，$R > C$ が望ましい．そこで，プロセス数が平方数の場合は R と C は同じ値に設定し，プロセス数が平方数でない場合は R の方が大きく，かつ R と C の差が最小になるように設定する．例えば，プロセス数が 8 の場合は $(R, C) = (4, 2)$ とする．

コードのコンパイルには富士通 MPI コンパイラ 1.2.24-03 を用いた．コンパイルオプションは “-Kfast -Kopenmp -Nfjomplib” である．

なお，本章では「富岳」を用いた評価を行うが，原稿執筆時点では共用前評価環境であるため，その評価結果は共用開始時の性能を保証するものではない．

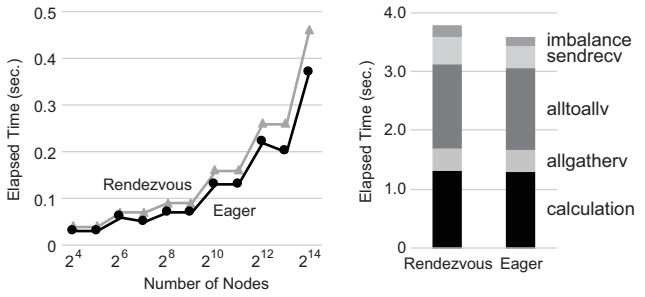
5.3 1 ノードにおけるプロセス数の最適化

本節では，最適な 1 ノードに割り当てるプロセス数（ppn：process per node）を調べる．評価に用いる ppn は，1，2，4 とした．それぞれの ppn におけるスレッド数は 48，24，12 である．プロセスの形状が $R \times C$ の場合のジョブ形状 (Y, X) は，1ppn の場合は (R, C)，2ppn の場合は $(R, C/2)$，4ppn の場合は $(R/2, C/2)$ とした．$(R, C) = (4, 4)$ の場合のプロセスマッピングの例を**図 14** に示す．図の四角はノードを表しており，**図 11a** では 8 ノード，**図 11b** では 4 ノードを用いている．

各 ppn の性能結果を**図 12** に示す．2ppn の場合の 16,384 ノードの結果は，システムの不具合により計測できなかった．1ppn の場合の 16,384 ノードの結果は 18,450GTEPS，4ppn の場合の結果は 20,970GTEPS であった．図 12 より，4ppn，2ppn，1ppn の順で性能は高いが，ノード数が多くなるにつれ，性能差は小さくなることがわかる．

次に，1ppn と 4ppn における BFS の各処理の時間割合を**図 13** に示す．図中の **calculation** はローカルの処理，**allgatherv**，**alltoallv**，**sendrecv** は表 4 と表 5 に示した通信時間，**imbarance** は各ステップの最後（while 文の最後）にバリア同期を行った際の同期待ち時間である．図 13 より，1ppn の方が 4ppn よりも通信時間の割合は小さいことがわかる．その原因は，1 ノードあたりの通信相手が 1ppn の方が少なく，また通信あたりのデータサイズが 1ppn の方が大きいからと考えられる．

本節の実験では，最大 16,384 ノードまでしか用いなかったが，最終的にはより多くのノードを用いた評価を行う．図 13 より，利用ノード数を増やすにつれ，通信時間の割合も増えていくことが予測できるため，以降の評価では通信時間の割合が小さい 1ppn で計測することにする．



(a) Elapsed time of **sendrecv**　(b) Breakdown in 2^{14} nodes

図 14: Comparison of Rendezvous and Eager

5.4 Eager 通信方式の利用

多くの MPI ライブラリの 1 対 1 通信では，Eager 通信方式と Rendezvous 通信方式の 2 種類が実装されている．Eager 通信方式は，送信側が受信側の状態にかかわらずバッファを介してメッセージを送信する非同期型の通信方式であり，小さいメッセージ通信に向いている．それに対し，Rendezvous 通信方式は，送信側が受信側のメッセージ格納先が確定するまでメッセージを送信しない同期型の通信方式であり，大きいメッセージ通信に向いている．

我々が開発した BFS においては，表 5 の **sendrecv** において 1 対 1 通信が行われる．5.3 節で行った実験において，その **sendrecv** に対する通信方式を調べた結果，すべての通信が Rendezvous 通信方式が用いられていることがわかった．「富岳」で提供されている富士通 MPI ライブラリの Eager 通信方式と Rendezvous 通信方式の切り替え閾値は，mpiexec コマンドのパラメータ **btl_tofu_eager_limit** で変更することができる．メモリの余裕がある場合は，**btl_tofu_eager_limit** の値を大きくすることで，Eager 通信方式の利用率を高めることができる．

そこで，本節では，すべての **sendrecv** に Eager 通信方式を用いた場合の性能評価を行う．その結果を図 14 に示す．比較として，図 14 では 5.3 節における 1ppn の結果を項目 “Rendezvous” として示している．**図 14a** は **sendrecv** の通信時間であり，**図 14b** は 16,384 ノード利用時の各処理の時間割合である．これらの結果より，Eager 通信方式を用いることにより，BFS の性能が向上することがわかる．Eager 通信方式を用いた場合の 16,384 ノードの結果は 19,496GTEPS であった．なお，図 14a において，測定値が階段状になっている理由は，表 5 にある通り，**sendrecv** の通信回数と通信サイズは C に比例し，C の値はノード数に対して飛び飛びに増えるからである（例えば，2^{12}，2^{13}，2^{14} ノード利用時の (R, C) の値は，それぞれ (64, 64)，(128, 64)，(128, 128) である）．

以降の評価では，すべての **sendrecv** が Eager 通信方式を利用するように **btl_tofu_eager_limit** の値を調整することとする．

5.5 ブーストモードとエコモードの利用

4 章で述べた通り，「富岳」ではジョブ単位で A64FX の周波数を 2.0GHz もしくは 2.2GH に指定できる．2.0GHz で動作する場合を “ノーマルモード”，2.2GHz で動作する場合を “ブーストモード” と呼ぶ．しかし，当然ではあるが，ブーストモードはノーマルモードよりも多くの電力を必要とする．そこで消費電力を下げるために，「富岳」では “エコモード” も用意されている．エコモード利用時は，A64FX が持つ 2 本の浮動小数点演算パイプラインが 1 本に制限されるとともに，その際の最大電力に合せた電力制御が行われる．BFS では浮動小数点演算は行わないため，エコモードを利用すると，性能に影響を与えずに消費電力を下げることが期待できる．そこで，本節では，ブーストモードおよびエコモードを利用した場合の BFS の性能と消費電力について評価する．ブーストモードとエコモードは直交した設定であるため，次の 4 つの組合せで評価を行う．

Normal　周波数は 2.0GHz で 2 本の浮動小数点演算パイプラインを利用（5.3 節と 5.4 節では，このモードを利用）

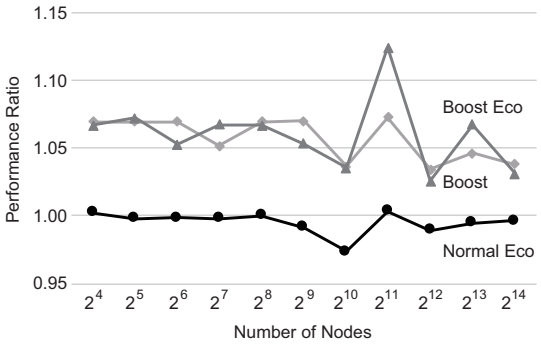
Boost　周波数は 2.2GHz で 2 本の浮動小数点演算パイプラインを利用

Normal Eco　周波数は 2.0GHz で 1 本の浮動小数点演算パイプラインを利用

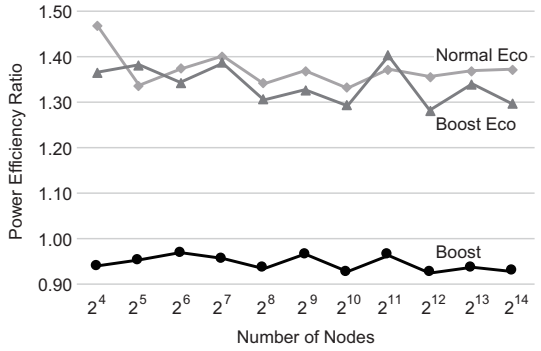
Boost Eco　周波数は 2.2GHz で 1 本の浮動小数点演算パイプラインを利用

なお，「富岳」では電力測定の方法がいくつか用意されており，大きく分けてユーザが行う方法と施設側で行う方法がある．ユーザが行う方法は，電力測定の範囲をプログラム中に専用の API を用いて指定し，ノード単位で電力を測定する．施設側で行う方法は，ジョブ全体をラック単位（「富岳」では 1 ラックに 384 ノードが格納されている）で計測するため，ラック内のすべてのノードを占有する必要がある．そのため，本節ではユーザが電力測定を行う方法を用いる．なお，ユーザが行う方法で計測するのは PSU（Power Supply Unit）から供給される直流電流であるのに対し，施設側で計測するのは PSU に供給している 200V の交流電流であるという違いがある．事前評価として，Normal で 3 ラックを占有して BFS の実行を行った結果，専用の API で計測した電力は 117kW であり，施設側で計測した電力は 126kW であった．この値の差は，AC/DC の変換ロスおよびノード電力に含まれないラック内の制御装置などの電力と考えられる [19]．

Normal の性能を 1.00 とした場合の各モードの性能比を **図 15a** に，Normal の電力効率（TEPS/W）を 1.00 とした場合の各モードの電力効率比を **図 15b** に示す．どちらの図においても，1.00 より高い値であれば，Normal よりも高い性能であることを意味する．まず，図 15a の結果より，ブーストモードに設定することで性能は 3〜12%ほ



(a) Comparison of performance



(b) Comparison of power efficiency

図 15: Comparison in each mode

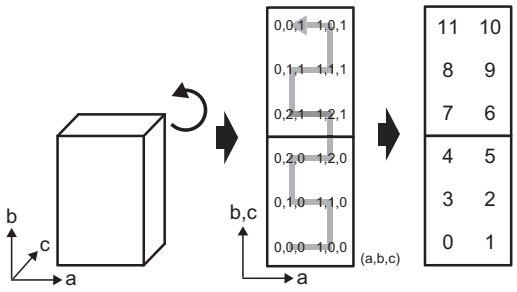


図 16: Process mapping for C dimension

ど高くなることがわかった．また，エコモードに設定しても，性能は変わらないことがわかった．次に，図 15b の結果より，エコモードに設定することで，電力効率は 28〜46%ほど高くなることがわかった．以上の結果より，性能と電力効率が共に高い Boost Eco が BFS に適していると言える．Boost Eco に設定した場合の 16,384 ノードの結果は，性能は 20,098GTEPS，消費電力は 1,369kW，電力効率は 14.69MTEPS/W であった．

5.6 6 次元プロセスマッピング

5.2 節で述べた通り，プロセス分割数である R と C は近い値が望ましい．しかしながら，「富岳」で指定できる 2 次元ジョブ形状の最大サイズは $YZc \times Xab$ であるため，「富岳」全系では 1,104 × 114 になり，R と C の差は 7.67 倍にもなる．そこで，Tofu-D の 6 次元ネットワークの任意の軸の組合せを R と C に設定できるプロセスマッピングを

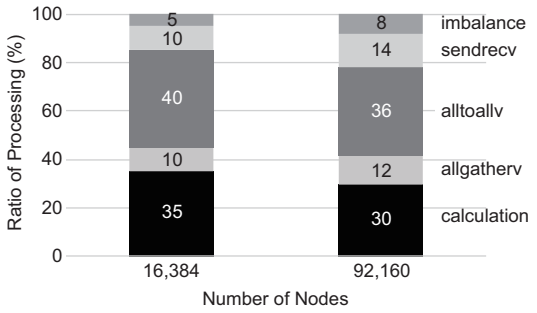


図 17: Time ratio of processing in 92,160 nodes

行う．例えば，「富岳」全系の場合，XY 軸に R を，$Zabc$ 軸に C を割り当てることで，552×288 のプロセスグリッドを作成でき，R と C の差は 1.92 倍になる．

C に対するプロセスマッピングでは，表 5 に示した **sendrecv** は隣接通信が適しているため，すべてのノードが隣接になるように工夫する．C に abc 軸（$2 \times 3 \times 2$）を割り当てる例を**図 16** に示す．まず，割り当てられた軸を 2 次元に展開する．横は最初の軸，縦は残りの軸を並べたものである．そして，図のように全プロセスが隣接になるように割り当てる．ただし，最初と最後のプロセス（図の例では 0 番と 11 番）を隣接にするため，下記の制約条件を満たす必要がある．(1) 2 次元に展開された縦方向の軸のサイズは偶数である．(2) 最後の軸の形状はトーラスもしくは a 軸か c 軸のどちらかである．R に対するプロセスマッピングは C のプロセスマッピングと基本的には同じであるが，上記の制約条件は必要ない．

原稿執筆時点では「富岳」は全系で利用できなかったため，本節では 92,160 ノードを用いた評価を行う．各軸の大きさは (X, Y, Z, a, b, c) = (20, 24, 24, 2, 3, 2) であるので，R と C の値はその差が最小になる (R, C) = (XY, $Zabc$) = (320, 288) に設定した．$SCALE$ は 40 に設定し，Boost Eco を設定した．その結果，性能は 70,980GTEPS，消費電力は 7,495kW，電力効率は 9.47MTEPS/W であった．また，この実験はラックを占有しているため，施設側でも消費電力の測定を行った．その消費電力と電力効率は，8,300kW と 8.55MTEPS/W であった [19].

本実験における各処理の時間割合を**図 17** に示す．比較のため，5.5 節で述べた Boost Eco を設定した 16,384 ノード利用時の結果も示している．92,160 ノードの方が 16,384 ノードのよりも **allgatherv** の割合が増えている．この理由は，16,384 ノードの R と C の値は共に 128 であるため，92,160 ノードの R と C と比較すると，R の増え幅が大きいからである．また，**sendrecv** の割合も増えている．16,384 ノードの問題サイズは 38 であるため，ノードあたりのグラフサイズは 92,160 ノードの方が小さい．そして，**sendrecv** は通信と計算のオーバラップを行っているが，計算量が相対的に減ったことにより，隠れていた通信時間が出てきたからと考えられる．

表 7: Graph500 list for June and November 2019 and June 2020

	June 2019		November 2019		June 2020	
	NAME	GTEPS	NAME	GTEPS	NAME	GTEPS
1st	**K computer**	**31,302**	Sunway TaihuLight	23,756	**Supercomputer Fugaku**	**70,980**
2nd	Sunway TaihuLight	23,756	Sequoia	23,751	Sunway TaihuLight	23,756
3rd	Sequoia	23,751	Mira	14,982	Summit	7,666
4th	Mira	14,982	Summit	7,666	SuperMUC-NG	6,279
5th	SuperMUC-NG	6,279	SuperMUC-NG	6,279	Cori	2,562

5.7 他のシステムとの比較

2019 年 6 月〜2020 年 6 月の Graph500 の 1〜5 位の結果を**表 7** に示す．2019 年 6 月の 1 位は「京」であり，これが最後のランキングであった．2019 年 11 月は，繰り上がりで Sunway TaihuLight [20] が 1 位になった．また，新マシンとして Summit [21] が 4 位にランクインしている．そして，5.6 節の結果により，2020 年 6 月に「富岳」が 1 位にランクインした．「富岳」の性能値は「京」の 2.27 倍，Sunway TaihuLight の 2.99 倍である．また，「京」の性能値は「富岳」を除いて破られていない．なお，Sequoia [22] と Mira [23] は運用終了したためランク外になった．

Graph500 ベンチマークにおける電力効率の性能を評価するランキングとして Green Graph500 がある [24]．Green Graph500 では，Graph500 にランクインしたシステムの中で，電力あたりに探索できる TEPS 値（TEPS/W）によってランキングが行われる．Green Graph500 には，問題サイズが 30 以上の BIG DATA 部門と，問題サイズが 29 以下の SMALL DATA 部門がある．しかしながら，$SCALE = 30$ は比較的小さい問題サイズであるため，BIG DATA 部門の上位は，ほぼ 1 ノードの結果で占められている．このことから，現在の Green Graph500 の規定は「富岳」のような大規模システムには適していないと言える．なお，全期間における投稿の中で問題サイズが 40 で電力効率が「富岳」を除いて最も高いのは Mira であり，その電力効率は 4.42MTEPS/W である．5.6 節の施設側での電力測定結果により，「富岳」の電力効率は 8.55MTEPS/W であるので，「富岳」の電力効率は Mira の 1.93 倍である．

6. まとめと今後の課題

本稿では，既存の Hybrid BFS をベースに改良を加えた BFS について述べ，「富岳」を用いた性能チューニングおよび評価を行った．問題サイズ 40（約 1.1 兆個の頂点と 17.6 兆本のエッジから構成される大規模グラフ）に対する BFS を「富岳」の 92,160 ノードを用いて性能評価を行った結果，70,980GTEPS を達成した．この結果により，2020 年 6 月の Graph500 で 1 位を獲得した．ハードウェア的な視点では，この結果は「富岳」が科学技術計算でよく用いられる規則的な計算だけでなく，不規則な計算が大半を占める BFS においても高い能力を持っていることを実証したものであり，「富岳」の汎用性の高さを示していると言える．

今後の課題としては，以下の点が挙げられる．(1)「富岳」の全系（158,976 ノード）を用いた評価を行う．(2) 「富岳」では，今回利用した富士通コンパイラ以外にも，clang などの様々なコンパイラも利用可能であるため，それらの利用についても検討する．(3) 詳細な性能モデリングを行い，ハードウェアと BFS の性能の関係について明らかにする．(4) 我々が開発した BFS を元に様々なグラフ処理を行うコード（SSSP も含む）を開発し，実データのグラフ処理を行うために「富岳」を活用していく．

Acknowledgements

本研究を進めるにあたり，「富岳」の運用をされている富士通株式会社のエンジニアの皆様および計算科学研究センターの運用技術部門の皆様にはお世話になりました．厚く御礼を申し上げます．また，フラグシップ 2020 プロジェクト（スーパーコンピュータ「富岳」）の石川裕リーダーにも心より感謝申し上げます．本研究の一部は，文部科学省「特定先端大型研究施設運営費等補助金（次世代超高速電子計算機システムの開発・整備等）」で実施された内容に基づくものであります．

参考文献

[1] Yan-Fen Da, Xing-Ming Zhao. A Survey on the Computational Approaches to Identify Drug Targets in the Postgenomic Era. *BioMed Research International*, p. 9, 2015. http://dx.doi.org/10.1155/2015/239654.

[2] Sylvain Brohee, Jacques van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC Bioinformatics*, 2006. https://doi.org/10.1186/1471-2105-7-488.

[3] Jordan Jakob et al. Extremely scalable spiking neuronal network simulation code: From laptops to exascale computers. *Frontiers in Neuroinformatics*, Vol. 12, p. 2, 2018.

[4] Peter Buhlmann et al., editor. *Handbook of Big Data*. Chapman and Hall/CRC, 2016.

[5] Richard C. Murphy, et al. Introducing the graph 500. Cray User’s Group, 2010.

[6] Graph500. https://graph500.org.

[7] Jure Leskovec et al. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, Vol. 11, No. 33, pp. 985–1042, 2010.

[8] Hiroyuki Miyazaki et al. Overview of the K computer. *FUJITSU SCIENTIFIC and TECHNICAL JOURNAL*, Vol. 48, No. 3, pp. 255–265, 2012.

[9] Supercomputer Fugaku. https://www.r-ccs.riken.jp/en/fugaku/project/.

[10] Scott Beamer et al. 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.
[11] Virat Agarwal et al. 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.
[12] S. Beamer et al. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 1618–1627, 2013.
[13] Koji Ueno et al. Efficient breadth-first search on massively parallel and distributed-memory machines. *Data Science and Engineering*, Vol. 2, pp. 22–35, 2016.
[14] Andy Yoo et al. 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.
[15] P.A. Boyle. The BlueGene/Q supercomputer. *PoS*, Vol. LATTICE2012, p. 020, 2012.
[16] RIKEN Center for Computational Science. https://www.r-ccs.riken.jp/en/.
[17] Y. Ajima et al. The tofu interconnect d. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 646–654, 2018.
[18] A64FX. https://github.com/fujitsu/A64FX.
[19] 児玉祐悦, 小田嶋哲哉, 有間英志, 佐藤三久. スーパーコンピュータ「富岳」における電力制御の性能評価. 情報処理学会研究報告 HPC 研究会, July 2020.
[20] Fu, Haohuan et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, Vol. 59, No. 7, Jun 2016.
[21] Summit. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.
[22] Barnes, Peter D. et al. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 327–336, 2013.
[23] S. Wallace et al. Measuring power consumption on ibm blue gene/q. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pp. 853–859, 2013.
[24] Green Graph500. https://graph500.org/?page_id=446.

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
9 ページ 右 19 行目	(X, Y, Z, a, b, c) = (20, 24, 24, 2, 3, 2)	(X, Y, Z, a, b, c) = (20, 16, 24, 2, 3, 2)