Vol.2014-HPC-143 No.11
2014/3/3

HPCシステムにおける汎用ネットワークトポロジ取得API nwtopoの提案

川島 崇裕[1]　安達 知也[1]　小田和 友仁[1]　住元 真司[1]　石川 裕[2]

概要：近年のHPCシステムでは，計算ノード数の増加につれ，計算ノードを接続するネットワークのトポロジも複雑・階層的になってきている．将来のHPCシステムではこの傾向が顕著になると予想される．しかし，性能的に最適なMPI集団通信のアルゴリズムやその通信パターンはネットワークトポロジに依存するものの，MPIライブラリがネットワークトポロジを検出するための汎用的なAPIは存在しない．本論文では，ネットワーク装置に依存せずにネットワークトポロジを問い合わせるためのAPIとして，nwtopoを提案する．nwtopoをスーパーコンピュータ「京」およびPRIMEHPC FX10のインターコネクトであるTofu用に実装した上で，ネットワークトポロジを考慮したMPI集団通信のアルゴリズムをnwtopoを使用して実装でき，その性能もTofu固有のAPIを使用した場合に比較して遜色ないことを示す．

nwtopo: A General Network Topology Querying API for HPC Systems

Abstract: As the increase of the number of computing nodes in recent HPC systems, the network topology connecting the nodes is becoming more complex and hierarchical. We believe this trend will continue for future HPC systems. MPI libraries should be aware of such complexity in order to decide optimal collective algorithms and optimal communication pattern in the selected algorithm. However, there are no standard and general APIs to detect network topologies. In this paper, we propose an API to query network topologies, named nwtopo. It absorbs the difference among network hardware and provides abstract topology information. We implement the API as a library for the Tofu interconnect, network hardware of the K computer and PRIMEHPC FX10, and also implement a topology-aware and network-hardware-dependent collective algorithm with the library as an experiment. The algorithm implementation is compared with an existing implementation which uses the Tofu-specific API. The evaluation result shows there is no significant performance degradation by using the general API, nwtopo.

1. はじめに

近年，HPC向け計算機システムのコア数は増加の一途をたどっている．LINPACKベンチマーク性能のランキングであるTOP500 [12] の2013年11月時点のリストでは，上位10システムのコア数は全て10万を超えており，100万コア以上を有するシステムも存在する．近年の大規模並列計算機のコア数増加には，単なる計算ノード数の増加だけでなく，計算ノードあたりのコア数の増加も寄与している．たとえば，Intel Xeon Phi Coprocessorはプロセッサあたりのコア数は60以上である．

このような，計算ノード数，計算ノードあたりのコア数の増加に伴い，それらを接続するノード間・コア間ネットワークは複雑なものとなっている．従来より，計算ノードの増加に従って，それらを1次元のクロスバで均一に接続することは現実的でなくなり，Fat-TreeやDragonflyといった，high-radixなルータで階層的に接続する間接網や，HypercubeやMesh，Torusといった，計算ノード同士を数珠つなぎに接続する直接網が提案されてきた．このようなネットワークでは，ノード間通信性能が均一でない場合があるほか，ノード間通信同士が干渉する可能性がある．近年ではさらに，ノード内のプロセッサ間の接続や，プロセッサ内のコア間の接続によっても通信のヘテロ性や干渉が生じる．

大規模並列計算では，特に集団通信において，ネットワークトポロジの複雑化を考慮に入れた通信の最適化が必要となる．集団通信では複数の通信が同期して動作するた

[1] 富士通株式会社
Fujitsu Limited
[2] 東京大学
The University of Tokyo

め，通信経路の衝突による輻輳などで生じた遅延が伝搬し全体性能の大幅な低下につながるためである．そのため，さまざまなネットワークトポロジに対して，数多くの集団通信アルゴリズムが提案されてきた [1,2,8,10,15]．

我々は，エクサフロップス級スーパコンピュータを目指すポストペタフロップススーパーコンピュータのためのシステムソフトウェアスタックの設計を進めている [16]．その取り組みの中で，最適な集団通信の在り方についても検討し，PC Cluster やスーパーコンピュータ「京」の後継等をテストベッドとして使用して評価していく．そのためには，様々なネットワーク装置と MPI 実装上で集団通信実装を評価する必要があり，その移植性の確保は重要な課題となっている．

しかし，従来，集団通信アルゴリズムはネットワーク装置と MPI 実装に合わせて，MPI ライブラリに組み込みで実装されてきた．そのため，ネットワーク装置と MPI 実装の組み合わせごとに別の実装として移植する必要があった．集団通信実装の移植性を高めるためには，汎用的なフォーマットでネットワークトポロジ情報を扱う必要がある．そこで，本報告ではネットワークトポロジ情報を取得するための汎用 API として nwtopo を提案する．nwtopo は，集団通信での利用だけではなく，一般にネットワークトポロジ情報を活用したい MPI ライブラリ実装者やアプリケーション実装者の利用も想定する．

2. 関連研究

既存のネットワークトポロジ情報取得 API としては，ノード内の構成情報を得るための hwloc [3] と，ノード間のネットワーク構成情報を得るための netloc [11] がある．

hwloc は，計算ノード内のコア，キャッシュ，メモリの階層情報を抽象化し，アーキテクチャ非依存の API で取得することができるライブラリである．MPICH [6] や Open MPI [5] といった広く使われている MPI ライブラリで，ノード内のプロセス配置 (CPU affinity) や動的メモリ割り当てポリシーの設定に利用されている．

netloc は hwloc の開発者らが開発した API であり，hwloc と組み合わせてノード内を含めた広義のネットワークトポロジを取得することができる．hwloc 同様，ノード間ネットワーク装置の違いを吸収し，ユーザは抽象化した形式でネットワークトポロジを取得できるようになっている．2013 年 11 月の SC13 での発表に合わせてベータ版 (Version 0.5) が公開された．2013 年現在の netloc の実装では，ネットワークトポロジをグラフで表現することから，超高並列環境ではメモリ使用量に対する懸念がある．また，複雑でヘテロ構成なネットワークも表現できるものの，それを使う集団通信の実装者は全体トポロジの構造を判定するのに全グラフを検索する必要があり，使い勝手が悪い．

我々の提案する nwtopo も hwloc と組み合わせて使用することを想定しており，nwtopo のインターフェース設計は hwloc を参考にしている．nwtopo では，さらに netloc での懸念点にも考慮している．nwtopo ではネットワークトポロジ自体をオブジェクトとして定義できるため，ノードとエッジの組み合わせによるグラフ表現よりも抽象度を増し，メモリ使用量を抑えることができる．また，オブジェクトの階層構造として複雑なヘテロ構成のトポロジもオブジェクトとして表現できる．よって，集団通信の実装者は nwtopo によりオブジェクトを取得することで，全体トポロジの判定が容易となる．

3. ネットワークトポロジ情報の活用

nwtopo の設計・実装について述べる前に，ネットワークトポロジ情報を活用する全体のイメージについて説明する．

3.1 ネットワークトポロジ情報の活用イメージ

一般に，メッセージサイズやプロセス数などの条件によって，最適な集団通信アルゴリズムは異なる．そのため，MPI ライブラリは複数の集団通信アルゴリズムを実装していて，実行時に最適なものを選択して使用するのが一般的である（使用する集団通信アルゴリズムをユーザが指定できるようなインターフェースを備えている MPI ライブラリも存在する [4]）．集団通信アルゴリズムには，多次元 Torus 向けに設計されたもの [7,9,13] など，特定のネットワークトポロジで高速に動作するものがある．そのため，最適アルゴリズムの選択には，ネットワークトポロジ情報を加味する必要がある．

また，ネットワークトポロジによらず使用できる汎用的な集団通信アルゴリズムであっても，通信の高速化においてネットワークトポロジ情報を利用することができる．集団通信では複数の通信が発生するため，プロセス間通信性能のヘテロ性や通信経路の衝突を加味した，通信経路および通信タイミングの最適化が重要である．この最適化には，通信範囲のノードとリンクの構成，リンクごとの通信性能差（レイテンシ，スループット）などの，ネットワークトポロジ情報が必要となる．ネットワークトポロジ情報を利用した具体的な通信経路の最適化の例は，3.2 節に示す．

以上 2 つの観点から，MPI ライブラリなどからネットワークトポロジ情報を取得するためのしくみが必要である．しかしながら，ネットワークトポロジ情報の取得方法は計算機環境によってまちまちである．たとえば，InfiniBand で接続された並列計算機の場合は，MAD (management datagram) パケットによりトポロジ情報を取得することができる．実装としては，ibnetdiscover コマンドや libibnetdisc が該当する．一方，Ethernet では，LLDP (Link Layer Discovery Protocol) パケットによってトポロジ検出を行う方法が標準化されている．また，並列計算機環境がクラスタ管理ミドルウェアを有する場合，システム全体のトポロ

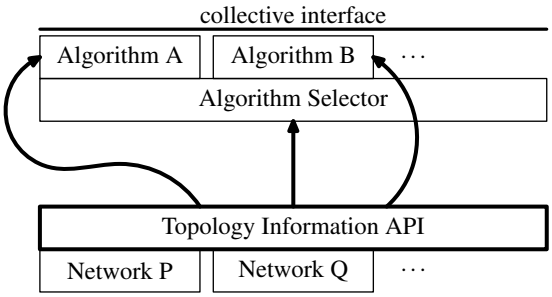


図 1 ネットワークトポロジ情報の活用イメージ

ジ情報や，プロセスが割り当てられた範囲のトポロジ情報を容易に取得できる場合がある．このようなネットワーク装置やクラスタ管理ミドルウェアごとのネットワークトポロジ情報の取得方法の違いは，MPI ライブラリなどの上位のライブラリやアプリケーションのポータビリティを低下させる原因となっている．ポータビリティ向上のためには，ネットワークトポロジ情報を取得するための汎用的な API が必要である．

図 1 に，汎用的なネットワークトポロジ情報取得 API を利用した集団通信最適化の模式図を示す．ネットワーク装置の提供者などが汎用 API の実装（ライブラリ）も提供することにより，MPI ライブラリなどネットワークトポロジ情報を利用したいライブラリやアプリケーションの実装者は，トポロジ情報を抽象化して扱うことができ，ネットワーク装置非依存のコーディングが可能となる．

3.2 ネットワークトポロジ情報の利用例

ネットワークトポロジ情報を利用する例として，集団通信のひとつである Bcast の binary tree アルゴリズムの最適化を挙げる．Bcast は，あるプロセス（root プロセスと呼ぶ）から他の全てのプロセスに同一のデータを送信する集団通信である．

図 2(a) に，16 プロセスでの binary tree アルゴリズムの通信パターンを示す．グラフの各ノードが 1 つのプロセスに対応しており，binary tree アルゴリズムでは，root プロセスから二分木状の通信パターンで，他の全てのプロセスにデータを転送していく．**図 2(b)** は，16 プロセスを 4×4 の 2 次元 Torus ネットワーク上に配置したものである．プロセスごとに 0 から 15 の番号がユニークに振られており，この番号（ランク番号と呼ぶ）で互いを識別する．以下，root プロセスはランク 0 であるとする．

集団通信の実装者は，グラフの各ノードにプロセスをマッピングする必要がある．ネットワークトポロジ情報がない場合は，何らかのルールで機械的にマッピングするしかない．**図 2(c)** は，二分木をランク番号順に単純に構成したものである．図 2(c) の二分木の通信パターンを図 2(b) 上に図示したものが，**図 2(d)** である．同一方向の経路の重なりは起きていないが，逆方向の経路の重なりがある．

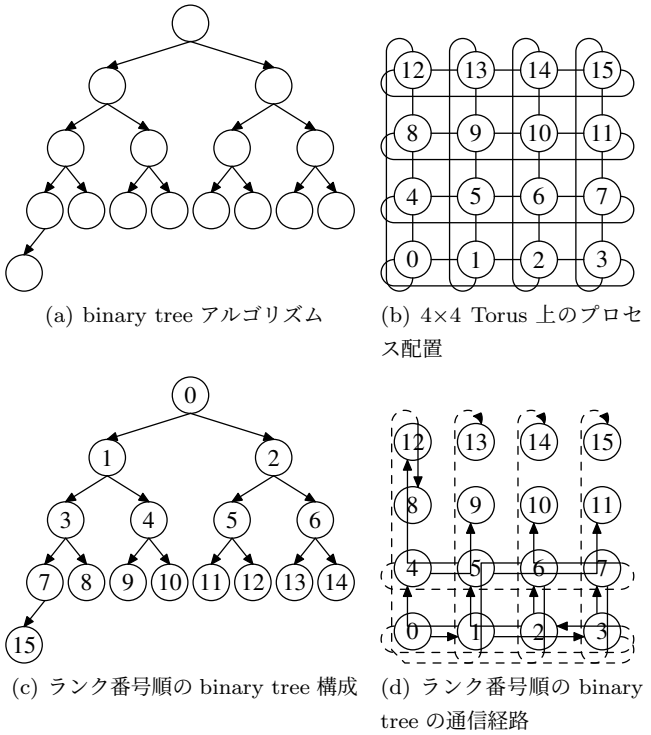


図 2 binary tree Bcast アルゴリズムの通信最適化の例

また，プロセス間の通信ホップ数は最大で 3 である．

一方，**図 2(e)** は，ネットワークトポロジ情報を利用して，通信経路が最適になるように二分木を構成したものである．図 2(e) の二分木の通信パターンを図 2(b) 上に図示したものが，**図 2(f)** である．通信経路を工夫することにより，経路の重なりはないほか，プロセス間の通信ホップ数は最大で 2 となり，図 2(c) の通信パターンよりも高速に動作することが期待される．

4. nwtopo の API 設計

4.1 API 要件と設計

これまでに述べたアプローチを実現するためには, nwtopo の API に以下の要件が挙げられる．

ネットワークトポロジの表現力 今日の HPC システムのネットワークトポロジで主流である多次元 Torus や Fat-Tree をネットワーク装置に依存せずに表現できるだけでなく，今後登場するであろう HPC システムのネットワークのトポロジを表現できる．

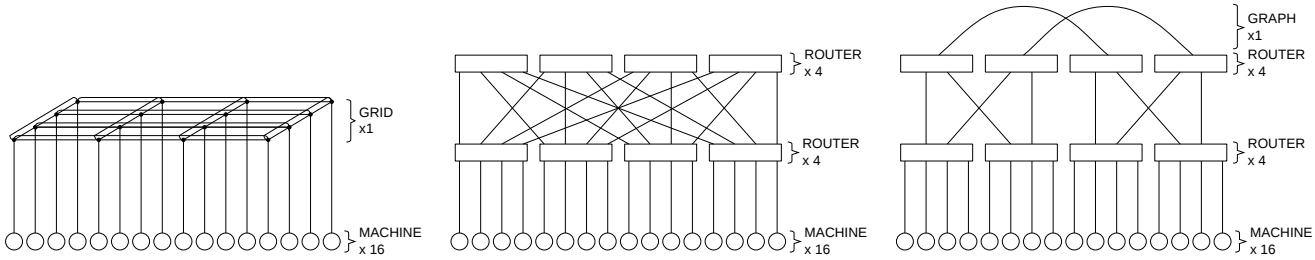


図 3 4×4 の 2 次元 Torus，Fat-Tree，Dragonfly のデータモデルの例

集団通信への適用性 集団通信アルゴリズムの選択やその通信最適化に必要な情報を取得できる．

大規模での省メモリ性 今日および今後登場するであろう HPC システムの規模で，計算ノードのメモリを圧迫するようなメモリを必要としない．

既存 API との親和性 既存の他の API と組み合わせて使用することができる．

4.1.1 ネットワークトポロジの表現力

任意のネットワークトポロジを表現するには，グラフによってノードとエッジを表現するのが一番単純である．しかし，ノードとエッジの情報を得るだけの API では，全体トポロジが多次元 Torus であることなどを知ることができず，集団通信の最適化に適用しにくい．一方，トポロジを多次元 Torus や Fat-Tree などに限定してしまうと，今後登場するであろう HPC システムのネットワークに対応できなくなってしまう．そこで，nwtopo では，現在の HPC システムのネットワークの論理的なトポロジ構成要素をオブジェクトとして定義し，オブジェクトの階層によって高度なトポロジを表現し，それでも表現できないトポロジはグラフによって表現できるように設計した．このデータモデルは 4.2 節で述べる．さらに，Tofu のようにネットワークの物理ビューと論理ビューが異なる場合 [17] は，トポロジ情報取得時のフラグによってどちらのビューの情報を取得するか選択できるようにしている．

4.1.2 集団通信への適用性

MPI の集団通信に適用するには，MPI_COMM_WORLD に含まれる計算ノードの集合だけでなく，MPI_Comm_split 関数などにより作成されたコミュニケータに含まれる計算ノードだけの部分集合を扱える必要がある．そのため，計算ノードの配列を指定して部分集合のトポロジ情報を取得する `nwtopo_topology_shrink` 関数を用意した．さらに，多次元 Torus では，各次元の軸の長さや計算ノードの相対的な位置といった情報が必要になる．オブジェクトの属性や関数によりこれらの情報を取得できるようにした．

4.1.3 大規模での省メモリ性

また，トポロジ情報の取得時に，必要のないオブジェクトを省略したり，`nwtopo_topology_shrink` 関数でオブジェクトを再利用したりするための，フラグを用意した．これにより，nwtopo の実装によってはメモリ使用量を抑えることができる．

4.1.4 既存 API との親和性

nwtopo の API の syntax と semantics を hwloc のものに近くすることで，ノード内の情報は hwloc，ノード間の情報は nwtopo で取得して，統一的に集団通信などをコーディングできるようにした．nwtopo の API の詳細は A.1 章に示す．

4.2 データモデル

nwtopo では，ネットワークトポロジをオブジェクトの集合の階層として表現する．オブジェクトの種類としては，以下のものがある．

machine 計算ノードを表す．

router ルーターを表す．InfiniBand のスイッチに相当する．

grid (mesh/torus) 子オブジェクトが Mesh や Torus で接続されていることを表す仮想的なオブジェクト．

full (full connection) 子オブジェクトが全対全で接続されていることを表す仮想的なオブジェクト．

graph 子オブジェクトが任意のエッジで接続されていることを表す仮想的なオブジェクト．

オブジェクトは他のオブジェクトと親子関係で接続される．例えば，InfiniBand の Fat-Tree における leaf スイッチは，親として他のスイッチと，子として計算ノードと接続されている．

さらに，それぞれのオブジェクトは以下の属性を持つ．

machine ホスト名，ネットワーク装置数．

grid (mesh/torus) 次元数，各次元の軸の長さ，各次元の周期性（Mesh か Torus か），など．

graph エッジの数と各エッジの両端のノード．

このデータモデルによる 4×4 の 2 次元 Torus，Fat-Tree，Dragonfly の例を，**図 3** に示す．4×4 の 2 次元 Torus では，Mesh/Torus であることを示す grid オブジェクトの子として machine オブジェクトが接続され，その grid オブジェクトの属性として，2 次元，4×4，Torus という情報が付与される．Fat-Tree では，それぞれの spine スイッチと leaf スイッチが router オブジェクトとして，計算ノードが machine オブジェクトとして，接続される．Dragonfly のように同一の階層のスイッチが接続される場合は，スイッ

```
nwtopo_topology_t topo;
nwtopo_obj_t me, *procs;
nwtopo_topology_init(&topo);
nwtopo_topology_load(topo);

me = nwtopo_topology_get_this_machine(topo);
ids = malloc(nprocs * sizeof(*ids));
procs = malloc(nprocs * sizeof(*procs));
allgather(&me->id, 1, MPI_UNSIGNED_LONG,
          ids, 1, MPI_UNSIGNED_LONG,
          comm);

for(i = 0; i < nprocs; i++){
  procs[i] = nwtopo_topology_get_object_by_id(
               topo, ids[i]);
}
```

図 4 コミュニケータ生成時のキャッシュ処理

```
nwtopo_obj_t robj = procs[root], oobj, nobj;
nwtopo_obj_t grid = me->parents[0];
int shifts[] = {-1, -1, -1};
int rc[3], mc[3], oc[3];

nwtopo_grid_get_coords(topo, grid, robj, rc);
nwtopo_grid_get_coords(topo, grid, me, mc);

nwtopo_grid_get_shifted_obj(topo, grid, robj,
                            shifts, &oobj);
nwtopo_grid_get_coords(topo, grid, oobj, oc);

for(id = 0; id < 3; id++){
  /* 送信方向 dirs[] を計算する処理 */
  ...

  for(i = 0; i < nsends; i++){
    nwtopo_grid_get_neighbor_obj(
      topo, grid, me, dirs[i], 1, &nobj);
  }
}
```

図 5 MPI_Bcast 呼び出し時の通信パターン決定処理

チの親として graph オブジェクトが接続され，その属性としてどのスイッチとどのスイッチが接続されているという情報が付与される．

4.3 nwtopo の使用例

本節では，6 章で評価向けの実装対象とする Trinaryx3 アルゴリズム [13] を例に，nwtopo の使用例を示す．なお，説明を簡単にするため，この例では 1 つの計算ノードに 1 つのプロセスが対応するものと仮定する．Trinaryx3 アルゴリズムは，3 次元 Torus 用の Bcast，Reduce，Allreduce に使用されるアルゴリズムであり，1 つのプロセスが最大で 3 つのプロセスに対してデータの送信を行う．送信先のプロセスは，すべて 3 次元 Torus 上で隣接しているプロセスである．

図 4 は，コミュニケータの生成時に，どのランク番号のプロセスがどの計算ノードに配置されているかを調べてキャッシュしておく処理である．まず，`nwtopo_topology_get_this_machine` 関数により自プロセスを実行している計算ノードのオブジェクト `me` を得る．次に，全計算ノードのオブジェクト ID を全プロセスで共有するために，`allgather` 関数（nwtopo が提供する API ではなく，MPI_Allgather またはそれに相当する MPI ライブラリ内部関数）を呼んでいる．同じオブジェクトを示すオブジェクト ID はどのプロセスでも同じであることが保証されているため，`nwtopo_topology_get_object_by_id` 関数によって各プロセスに対応するオブジェクトを取得することができる．

MPI_Bcast 関数が呼ばれたときには，root プロセス，root プロセスから最も遠いプロセス，自プロセスの座標情報から，自プロセスがデータを送信する先のプロセス `nobj` を計算する．この処理の nwtopo 使用部分を **図 5** に示す．この例では，3 次元 Torus 向けのアルゴリズムであるため，mesh/torus 固有の API を使用して，`nwtopo_grid_get_shifted_obj` 関数により root プロセスから $(-1,-1,-1)$ だけ移動した位置にいるプロセスを取得し，`nwtopo_grid_get_neighbor_obj` 関数により，特定の方向に 1 だけ移動した位置にいるプロセスを取得している．

5. nwtopo の実装

nwtopo の API を検証するため，nwtopo をスーパーコンピュータ「京」および PRIMEHPC FX10 のインターコネクトである Tofu 用に実装した．

Tofu ネットワークの物理的な接続は，XYZABC の 6 軸から構成される 6 次元の Mesh/Torus であり，軸によって端がつながっているかどうか（Mesh か Torus か）が異なる．しかし，複数の軸を論理的に組み合わせ，ユーザには 3 次元または 2 次元・1 次元の Torus として見えるようにしている．そのため，nwtopo の実装でも，物理ビューのフラグを設定すれば 6 次元 Mesh/Torus，設定しなければ 3 次元・2 次元・1 次元の Torus として，情報が得られるようにした．

スーパーコンピュータ「京」の最大構成である 82,944 ノードのトポロジ情報を取得した場合のデータのメモリ使用量の理論値は，およそ 10MiB になる．しかし，各計算ノードの情報が不要で全体のトポロジ情報だけ必要な場合は，フラグ `NWTOPO_TOPOLOGY_FLAG_NO_MACHINES` を指定することにより，データのメモリ使用量は 100B 程度に削減される．また，MPI_Comm_split 関数などにより MPI_COMM_WORLD と同じ大きさのコミュニケータを作成した場合は，フラグ `NWTOPO_TOPOLOGY_FLAG_REUSE_OBJECTS` を指定すれば，追加で使用するメモリ量はおよそ 1MiB になる．

表 1 測定環境

CPU	SPARC64 IXfx
CPU 周波数	1.848GHz
メモリ	64GB/ノード
インターコネクト	Tofu インターコネクト
ネットワークスループット	5GB/s× 双方向 ×4 リンク
使用したノード形状	4×6×16 (3 次元 Torus)

表 2 通信パターン計算記述行数

	初期化	通信パターン計算
nwtopo	37	147
MPI ライブラリ	227	245

6. nwtopo の評価

本章では, スーパーコンピュータ「京」および PRIMEHPC FX10 の MPI ライブラリに実装されている Trinaryx3 アルゴリズム [13] を，nwtopo を用いて移植実装することで，評価を行う．nwtopo はアルゴリズム選択ではなく Trinaryx3 アルゴリズムの通信パターンの計算に用いる．

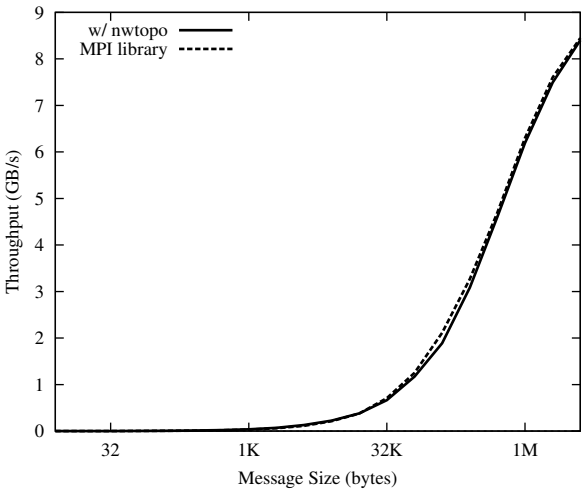
評価は以下の手順で行う．まず，通信パターン計算部分の記述量について，既存実装との比較を行い，nwtopo を使用することで記述量が減ることを確かめる．次に，既存実装との性能比較を行い，性能の傾向から，ネットワーク装置に依存しない記述で同等の集団通信アルゴリズムが実装できていることを確かめる．最後に，nwtopo のメモリ使用量をシミュレーションにより評価し，超大規模並列でのメモリ使用量を見積もる．

評価には，富士通沼津工場に設置された PRIMEHPC FX10 を使用した．諸元は**表 1** のとおりである．性能評価には最大 4×6×16 の 384 ノードを使用し，メモリ使用量の評価は，1 ノード上でシミュレーションを行った．

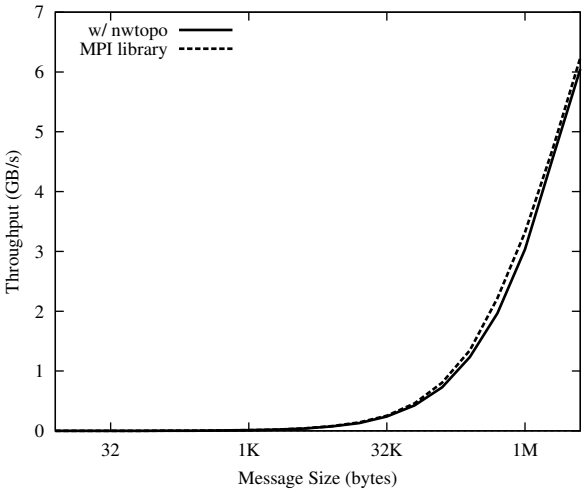
6.1 nwtopo を使用した場合のコード記述量

表 2 に，Trinaryx3 アルゴリズムの通信パターンの計算に関連するコード行数を，nwtopo を使用して記述した場合と移植元の MPI ライブラリのそれぞれについて示す．一度だけ呼ばれる初期化部分と，MPI_Bcast の実行時に通信パターンを計算する部分とにわけて計上している．

移植元の MPI ライブラリでは, システムが提供する Tofu インターコネクト専用のライブラリ（以下 Tofu ライブラリ）を使用して，その結果を元に抽象化したトポロジ情報を求めている．そのため，特に初期化部分の記述量が多くなっている．通信パターンの計算においても，隣接座標にいるプロセスを求める処理などで Tofu ライブラリを使用しており，記述量が増えている．nwtopo を使用した場合は，それらの処理は全て nwtopo ライブラリの内部に隠蔽されており，集団通信アルゴリズムの記述が容易になっていることがわかる．



(a) 12 ノード



(b) 384 ノード

図 6 MPI_Bcast スループット

6.2 nwtopo を使用した集団通信アルゴリズム性能

本報告の評価では，nwtopo を用いたアルゴリズム実装を容易にするため，通信部には PRIMEHPC FX10 の MPI ライブラリが提供する拡張 RDMA インターフェースを使用した．拡張 RDMA インターフェースは，MPI アプリケーションレベルで低オーバーヘッドの RDMA 通信を記述可能とする API であり，Tofu インターコネクト以外のネットワークにも対応できるよう抽象化されている．nwtopo と拡張 RDMA インターフェースにより，ネットワーク装置非依存のアルゴリズム記述が実現されている．移植元の MPI ライブラリは通信部に Tofu ライブラリを使用しているため，通信部の違いによって性能の違いが発生する可能性があるが，本報告の評価は，性能の傾向を比較することで，同一アルゴリズムが nwtopo を用いて記述できていることを確かめるのが目的である．

性能測定には Intel MPI Benchmarks を使用した．2×3×2 の 12 ノードでの MPI_Bcast スループットを**図 6(a)** に，4×6×16 の 384 ノードでの MPI_Bcast スループットを

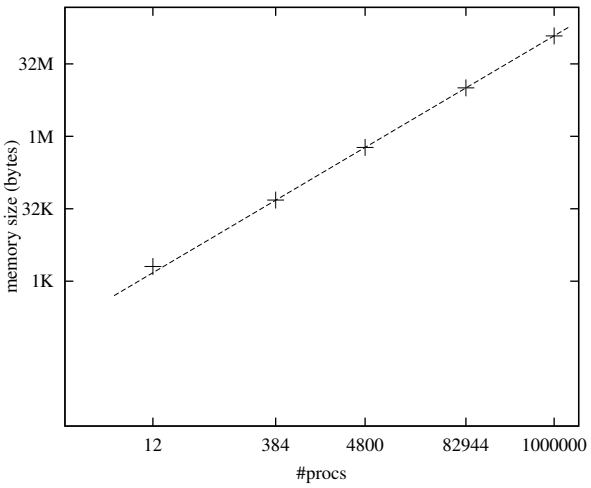


図 7 nwtopo メモリ使用量

図 6(b) に示す．どちらの形状でも，nwtopo を使用して記述したアルゴリズムと，移植元の MPI ライブラリの同一アルゴリズムで，性能の傾向は一致しており，問題なくアルゴリズムが移植できていることがわかる．nwtopo を使用して記述した場合に見られる若干の性能低下は，nwtopo 関数群の呼び出し（転送メッセージサイズによらず固定コスト）によるものではなく，システムが提供する Tofu インターコネクト専用通信ライブラリの代わりに拡張 RDMA インターフェースを利用して抽象化したことによるオーバーヘッドによるものである．

6.3 nwtopo のメモリ使用量

プロトタイプ実装の nwtopo では，外部からプロセス数とそのプロセスが割り当てられた形状の情報を与えることができる．この機能を用いて，全体のトポロジ情報を取得して Trinaryx3 アルゴリズムの通信パターンを計算するまでの部分の nwtopo のメモリ使用量について，プロセス数を変更しながら評価を行った．nwtopo ライブラリから取得されるメモリ量の測定には，DMATP-MPI [14] を用いた．形状とプロセス数としては，2×3×2（12 ノード），4×6×16（384 ノード），20×15×16（4,800 ノード），48×54×32 (82,944 ノード)，100×100×100 (1,000,000 ノード）の 5 種類の条件で計測した．

メモリ使用量の測定結果を**図 7** に示す．京の最大構成（82,944 ノード）でのメモリ使用量は 10.1MiB であり，5 章で述べた理論値と合致している．また，メモリ使用量はプロセス数に対して線形にスケールしていることがわかる．

7. まとめ

本報告では，ネットワーク装置に依存せずにネットワークトポロジを問い合わせるための API である nwtopo を提案した．nwtopo は，様々なネットワークを表現でき，集団通信への適用性，大規模での省メモリ性，既存 API との親和性を考慮した設計となっている．MPI ライブラリなどネットワークトポロジ情報を利用したいライブラリやアプリケーションは，抽象化されたトポロジ情報を用いてポータビリティの高い実装が可能となる．

PRIMEHPC FX10 上での評価の結果，nwtopo を用いて記述した集団通信実装は，MPI ライブラリの同一アルゴリズムの実装と比較して，同程度の通信性能を実現しながらコード行数は 40%以下になっており，少ない記述量でネットワーク装置に依存しない実装が可能であることが示された．また，トポロジ情報取得時の nwtopo のメモリ使用量を同環境で測定した結果，机上の計算通り，ノード数に対し線型にスケールし，見積もり可能であることを確認した．

今後の課題としては以下が考えられる．本報告で示したデータ構造では，リンクのバンド幅や故障情報など，リンクに関する情報を表現することができない．router オブジェクトや grid オブジェクトの属性として参照できるような拡張が必要である．また，クラスタ管理ミドルウェアからの情報取得を考えると，計算ノード外から情報取得できるようなしくみも必要である．

本報告では，3 次元 Torus 上で単一アルゴリズムの移植により評価を行った．Fat-Tree など他のネットワーク形状での評価や，複数アルゴリズムの選択での評価は，今後の課題である．また，nwtopo の実用性を示すには，より多様なネットワーク装置上で nwtopo ライブラリが実装され，使用されるべきである．集団通信の実装者が nwtopo の API を使用してアルゴリズムを記述し，その実装を共有できるようなコミュニティの形成が必要と考えられる．

謝辞 本研究の一部は，文部科学省「将来の HPCI システムのあり方の調査研究」のなかの課題名「レイテンシコアの高度化・高効率化による将来の HPCI システムに関する調査研究」によるものである．

参考文献

[1] Adachi, T., Shida, N., Miura, K., Sumimoto, S., Uno, A., Kurokawa, M., Shoji, F. and Yokokawa, M.: The Design of Ultra Scalable MPI Collective Communication on the K Computer, *Computer Science – Research and Development*, Vol. 28, No. 2–3, pp. 147–155 (2013).

[2] Almási, G., Archer, C., Erway, C. C., Heidelberger, P., Martorell, X., Moreira, J. E., Steinmacher-Burow, B. D. and Zheng, Y.: Optimization of MPI Collective Communication on BlueGene/L Systems, *Proc. of ICS 2005*, pp. 253–262 (2005).

[3] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S. and Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, *Proc. of PDP 2010*, pp. 180–186 (2010).

[4] Fagg, G. E., Bosilca, G., Pješivac-grbović, J., Angskun, T. and Dongarra, J. J.: Tuned: An Open MPI Collective Communications Component, *Distributed and Parallel Systems*, Springer, pp. 65–72 (2007).

[5] Graham, R. L., Shipman, G. M., Barrett, B. W., Cas-

tain, R. H., Bosilca, G. and Lumsdaine, A.: Open MPI: A High-Performance, Heterogeneous MPI, *Proc. of HeteroPar 2006* (2006).
[6] Gropp, W., Lusk, E. L., Doss, N. E. and Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, *Parallel Computing*, Vol. 22, No. 6, pp. 789–828 (1996).
[7] Jain, N. and Sabharwal, Y.: Optimal Bucket Algorithms for Large MPI Collectives on Torus Interconnects, *Proc. of ICS 2010*, pp. 27–36 (2010).
[8] Johnsson, S. L. and Ho, C.-T.: Optimum Broadcasting and Personalized Communication in Hypercubes, *IEEE Transactions on Computers*, Vol. 38, No. 9, pp. 1249–1268 (1989).
[9] Kumar, S., Sabharwal, Y., Garg, R. and Heidelberger, P.: Optimization of All-to-All Communication on the Blue Gene/L Supercomputer, *Proc. of ICPP '08*, pp. 320–329 (2008).
[10] Mamidala, A. R., Kumar, R., De, D. and Panda, D. K.: MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics, *Proc. of CCGrid 2008*, pp. 130–137 (2008).
[11] netloc: `http://www.open-mpi.org/projects/netloc/`.
[12] TOP500: `http://www.top500.org/`.
[13] 松本　幸，安達知也，田中　稔，住元真司，曽我武史，南里豪志，宇野篤也，黒川原佳，庄司文由，横川三津夫：MPI_Allreduce の「京」上での実装と評価，情報処理学会研究報告，Vol. 2011-HPC-132, No. 6 (2011).
[14] 秋元秀行, 安島雄一郎, 安達知也, 岡本高幸, 三浦健一, 住元真司：DMATP-MPI: MPI 向け動的メモリ割当分析ツール，情報処理学会研究報告，Vol. 2013-HPC-138, No. 14 (2013).
[15] 成瀬　彰，中島耕太，住元真司，久門耕一：マルチコア PC クラスタ向け All-to-all 通信アルゴリズムの提案と評価，情報処理学会論文誌コンピューティングシステム，Vol. 3, No. 3, pp. 166–177 (2010).
[16] 石川　裕, 堀　敦史, Gerofi, B., 高木将通, 島田明男, 清水正明, 佐伯裕治, 白沢智輝, 中村　豪, 住元真司, 小田和友仁：次世代高性能並列計算機のためのシステムソフトウェアスタック，情報処理学会研究報告，Vol. 2013-OS-125, No. 3, pp. 1–8 (2013).
[17] 安島雄一郎, 井上智宏, 平本新哉, 清水俊幸：スーパーコンピュータ「京」のインターコネクト Tofu，雑誌 FUJITSU, Vol. 63, No. 3, pp. 260–264 (2012).

付　　録

A.1 nwtopo の API

nwtopo の API を以下に示す.

復帰値の型が `int` である関数では，正常時には 0 を，異常時には −1 を返す.

A.1.1 構造体・列挙型など

```
typedef struct nwtopo_topology *
        nwtopo_topology_t;
```

nwtopo のハンドル.

```
typedef enum {
  NWTOPO_OBJ_MACHINE,
  NWTOPO_OBJ_ROUTER,
  NWTOPO_OBJ_GRID,
  NWTOPO_OBJ_FULL,
  NWTOPO_OBJ_MESH,
  NWTOPO_OBJ_MAX
} nwtopo_obj_type_t;
```

オブジェクトの種類.

```
struct nwtopo_obj {
  unsigned long             id;
  nwtopo_obj_type_t         type;
  unsigned                  height;
  unsigned                  num_parents;
  unsigned                  num_children;
  struct nwtopo_obj       **parents;
  struct nwtopo_obj       **children;
  union nwtopo_obj_attr_u *attr;
  void                      *userdata;
};
typedef struct nwtopo_obj *
        nwtopo_obj_t;
```

各オブジェクト．`id` は一意の ID 値, `parents`, `children` はそれぞれ `num_parents`, `num_children` 個の配列, `userdata` は使用者が任意に設定して良い変数である.

```
union nwtopo_obj_attr_u {
  struct {
    char    *hostname;
    unsigned num_nics;
  } machine;
  struct {
    int dummy;
  } router;
  struct {
    bool          dense;
    unsigned      num_dimensions;
    unsigned     *dimensions;
    unsigned     *reference_coordinates;
    bool         *periodicities;
    nwtopo_obj_t *map;
  } grid;
  struct {
    int dummy;
  } full;
  struct {
    unsigned            num_edges;
    nwtopo_mesh_edge_t *edges;
  } mesh;
};
```

オブジェクトの属性.

```
struct nwtopo_mesh_edge {
```

```
  nwtopo_obj_t nodes[2];
};
typedef struct nwtopo_mesh_edge *
        nwtopo_mesh_edge_t;
```

Graph における各エッジ.

```
enum nwtopo_topology_flags_e {
  NWTOPO_TOPOLOGY_FLAG_PHYSICAL_VIEW = (1<<0),
  NWTOPO_TOPOLOGY_FLAG_WHOLE_SYSTEM  = (1<<1),
  NWTOPO_TOPOLOGY_FLAG_NO_MACHINES   = (1<<2),
  NWTOPO_TOPOLOGY_FLAG_NO_HOSTNAMES  = (1<<3),
  NWTOPO_TOPOLOGY_FLAG_REUSE_OBJECTS = (1<<4)
};
```

nwtopo_topology_set_flags 関数に指定するフラグ.

A.1.2 トポロジ情報の生成・破棄

```
int nwtopo_topology_init(
  nwtopo_topology_t *topologyp);
```

新たな nwtopo_topology_t オブジェクトを生成する. nwtopo_topology_load 関数を呼ぶまではネットワークトポロジ情報を取得することはできない.

```
int nwtopo_topology_load(
  nwtopo_topology_t topology);
```

フラグに基づいてネットワークトポロジ情報を分析して nwtopo_topology_t オブジェクトに設定する.

```
int nwtopo_topology_shrink(
  nwtopo_topology_t new_topology,
  nwtopo_topology_t base_topology,
  unsigned          num_machines,
  nwtopo_obj_t     *machines);
```

base_topology に含まれる num_machines 個の計算ノードの配列 machines からなる部分集合を作成する.

```
void nwtopo_topology_destroy(
  nwtopo_topology_t topology);
```

nwtopo_topology_t オブジェクトを破棄する.

A.1.3 トポロジ検出の設定

```
unsigned long nwtopo_topology_get_flags(
  nwtopo_topology_t topology);
```

nwtopo_topology_set_flags 関数で設定したフラグを取得する.

```
int nwtopo_topology_set_flags(
  nwtopo_topology_t topology,
  unsigned long     flags);
```

ネットワークトポロジの検出条件を設定する. flags は nwtopo_topology_flags_e 列挙子の論理和である.

A.1.4 トポロジ情報の取得

```
unsigned nwtopo_topology_get_max_height(
  nwtopo_topology_t topology);
```

ネットワークトポロジが階層状になっているときの最上位のオブジェクトの高さを取得する. 高さは計算ノードが 0 でその 1 つ上の階層が 1 である.

```
nwtopo_obj_t nwtopo_topology_get_object_by_id(
  nwtopo_topology_t topology,
  unsigned long     id);
```

特定の ID を持つオブジェクトへの参照を取得する.

```
nwtopo_obj_t nwtopo_topology_get_this_machine(
  nwtopo_topology_t topology);
```

このプロセスを実行している計算ノードのオブジェクトへの参照を取得する.

A.1.5 Mesh/Torus 固有の情報の取得

```
int nwtopo_grid_get_obj(
  nwtopo_topology_t topology,
  nwtopo_obj_t      grid,
  int              *coords,
  nwtopo_obj_t     *objp);
```

Mesh/Torus grid 上の特定の座標 coords（配列）に位置するオブジェクトへの参照を取得する.

```
int nwtopo_grid_get_shifted_obj(
  nwtopo_topology_t topology,
  nwtopo_obj_t      grid,
  nwtopo_obj_t      base,
  int              *shifts,
  nwtopo_obj_t     *objp);
```

Mesh/Torus grid 上でオブジェクト base から shifts（配列）だけ移動したところに位置するオブジェクトへの参照を取得する.

```
int nwtopo_grid_get_neighbor_obj(
  nwtopo_topology_t topology,
  nwtopo_obj_t      grid,
  nwtopo_obj_t      base,
  unsigned          direction,
  int               shift,
  nwtopo_obj_t     *objp);
```

Mesh/Torus grid 上でオブジェクト base から direction 方向に shift だけ移動したところに位置するオブジェクトへの参照を取得する.

```
int nwtopo_grid_get_coords(
  nwtopo_topology_t topology,
  nwtopo_obj_t      grid,
  nwtopo_obj_t      obj,
  int              *coords);
```

Mesh/Torus grid 上でオブジェクト obj の座標を取得する.