Vol.2019-HPC-168 No.3
2019/3/5

Open Fabrics Interfacesオフロード APIを用いたMPI永続型集団通信の設計と評価

森江 善之[1,a)] 畑中 正行[1] 高木 将通[1] 堀 敦史[1] 石川 裕[1]

概要：Open Fabrics Intercaces は通信オフロード API としてメッセージ受信数で通信処理を発行する機能が提供されている。本機能を用いて、永続型集団通信の通信アルゴリズムをネットワークオフロードする実装方式を提案する。通信オフロードを利用した集団通信実装の多くは、1 対 1 通信をネットワークオフロードする機構を実装し、集団通信アルゴリズムはこの 1 対 1 通信機構を用いて実装されている。すなわち、集団通信アルゴリズムがオフロードされているものではない。提案する実装方式と 1 対 1 通信オフロードベースの実装方式の必要とするハードウェア資源量について比較する。Barrier や Allgather などの集団通信では、提案方式では O(1) のハードウェア資源量で済むが、1 対 1 通信オフロードベースの実装方式では、$O(\log_2 N)$ のハードウェア通信資源が必要となることを示す。

1. はじめに

大規模スーパーコンピュータにおける集団通信性能高速化のひとつして、新しい非同期集団通信関数である永続型集団通信の導入が予定されている [1]。永続型集団通信は、初期化処理を準備関数として独立させ、集団通信の各インスタンスにおいて初期化処理を省略可能とするもので、高速化への貢献が考えられる。

現在の大規模並列計算機は、高速かつ高機能なインターコネクトで結合されており、これらのインターコネクトは通信処理をオフロードする機能がある [2][3]。インターコネクトのオフロード機能を用いて 1 対 1(send/recv) 通信機能を実現し、この機能を組み合わせて集団通信機能を実現する方法がある [4]。本実装では集団通信アルゴリズムを実現するために、例えば、Barrier, Allgather では、$O(\log_2 N)$ のハードウェア通信資源が必要となる。

send/recv 通信オフロード機能を用いて永続型集団通信を実現する場合、準備関数内でオフロードのための事前準備が行われ、必要なハードウェア通信資源が割り当てられることになる。永続型集団通信は同じパラメタの通信が繰り返し何度も呼ばれることを想定しており、通常、このような資源は集団通信インスタンスが消滅するまで保存される。このため、同時に複数の永続型集団通信インスタンスが作られるとハードウェア通信資源枯渇の可能性が生じる。

[1] 理化学研究所 計算科学研究センター
Riken Center for Computational Science (R-CCS), kobe, hyogo, 650-0047, Japan
[a)] yoshiyuki.morie@riken.jp

MPICH や OpenMPI など多くの MPI 通信ライブラリの実装は階層構造になっており、デバイス独立部と依存部にわかれている。例えば、MPICH においてはデバイス依存部として、Open Fabrics Interfaces (OFI)[5] や Portals4[6] が定義されている。Tofu2[2] のように固有の API を有するインターコネクトのオフロード機能をこのような通信ライブラリに適用するには独自のデバイス依存部を実装するか既存デバイス依存部に合わせた実装のどちらかを選択することになる。前者は開発コストと保守性の面で問題がある。そこで、本稿では、Tofu2 等でも適用可能な OFI (libfabric) を基に OFI が提供する `fi_trigger` と呼ばれるオフロード機能を用いて Broadcast, Barrier, Allgather を例に永続型集団通信アルゴリズムをオフロードできることを示し、これにより必要とされるハードウェア通信資源量は O(1) になることを示す。

本稿は以下のような構成となる。まず、第 3 節では、関連研究について述べる。また、第 4 節で libfabric オフロード API を用いた永続型集団通信を設計する。第 5 節で設計した永続型集団通信の資源量と send/recv 通信オフロード機能を用いた永続型集団通信の資源量を比較する。最後にまとめと今後の課題を述べる。

2. 背景

2.1 永続型集団通信

永続型集団通信は、初期化関数である MPI_(Bcast/Allgather/...)_init、開始関数の MPI_Start(), 完了確認関数の MPI_Wait()、資源解放関数 MPI_Request_free() の 4 個

の関数で構成され、MPI_Start() による永続型集団通信の発行、MPI_Wait() で完了確認を行うインターフェースは非同期な動作が可能となる。永続型集団通信の実行例として図 1 に疑似コードを示す。

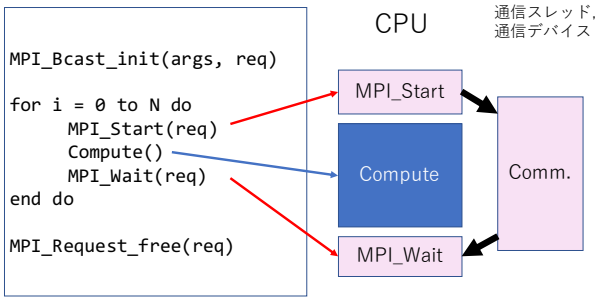

図 1 永続型集団通信の動作概要

永続型集団通信は通信デバイス等へのハードウェアによるオフロード実装により通信の高速化が図れる可能性がある。準備関数で集団通信に必要なハードウェア資源の占有や必要な初期化を行う。通信開始関数においては予め用意したハードウェア資源を用いて集団通信を行う。CPU の介在なしに通信処理を進めることが出来るならば通信と計算のオーバラップも可能となる。しかし、このような実装ではハードウェア資源は資源開放関数が呼ばれるまで専有されるため、大量の永続型集団通信インスタンスが作られると、限られたハードウェア資源が枯渇する可能性がある。

2.2 libfabric トリガー型操作

Deferred Work (DW) 要求は、libfabric `fi_trigger(3)` API の一つで、`fi_deferred_work` 構造体で表現される。この構造体は、発報条件と発報時の行動を記述する。これを図 2 に示す。

```
struct fi_deferred_work { ...
  uint64_t threshold;
  struct fid_cntr * triggering_cntr; ...
  enum fi_tringer_op op_type;
  union { ...
  } op;
};
```

図 2 Deferred Work (DW) 要求の構造

この要求により、フィールド *triggering_cntr* が指すカウンタの値が *threshold* 以上になったとき、*op_type* および *op* で指定された操作が実行されるよう通信スケジューラに依頼することができる。

この要求は、複数の水位スイッチが取付られた貯水槽でモデル化可能である。以降、このモデルを「貯水槽 (Water Tank) モデル」と呼ぶ。これを図 3 に示す。

このモデルでは、libfabric `fi_cntr(3)` の挙動を貯水槽 (Water Tank) に見立てる。このモデルと DW 要求の対応関係を表 1 に示す。貯水槽の水位 (Water Level) は libfabric のカウンタ値に相当し、`fi_cntr_read()` によって取得

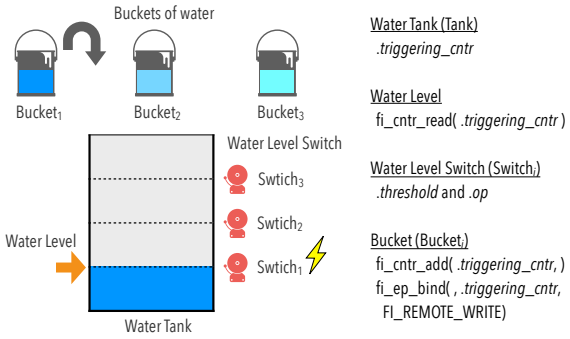

図 3 貯水槽モデルの概要

できる。水位スイッチ (Water Level Switch) はある水位に達したら ON になるような単純なスイッチで、DW 要求の *threshold* と *op* (ならびに *op_type*) に相当する。

表 1 Deferred Work (DW) 要求とモデルの対応関係

略号	DW フィールド名	備考
Ct	.triggering_cntr	貯水槽 *Tank*
Th	.threshold	*Tank* に設置する *Switch* の設定水位
Ot	.op_type	*Switch* 発報時の操作タイプ
Op	.op	*Switch* 発報時の操作パラメタ

水くみバケツ (Buckets of water) の水はカウンタの増分値である。決められた量の水の入ったバケツ *Bucket* から貯水槽への注水以外に、貯水槽の水位を変更できないものとし (単調増加)、それぞれの注水自体は一瞬で完了するものとする (アトミック)。この注水は、リモートからの `FI_OP_WRITE` の受信 (+1) や自側での `fi_cntr_add` 呼び出し (+n) の処理完了を示すイベントとして発生する。

このモデルは実際、BXI (Bull eXascale Interconnect) [3] や Tofu2 Session-Mode [2] のような最近のオフロード可能なインターコネクトハードウェアに通底するモデルである。以降の節では、このシンプルかつハードウェア寄りのモデルで、永続集団通信のオフローディングを再検討してゆく。

2.3 オフロード可能なインターコネクトの資源量

BXI は native で Portals 4 仕様をサポートする。libfabric の `fi_trigger` は Portals 4 トリガー型操作をほぼ忠実に抽象化したもので、Portals 4 のタグマッチングを除き、トリガー型操作関連関数間に対応関係がある。Portals 4 ユーザは、下位のインターコネクトの資源量を `ptl_ni_limits_t` 構造体を介し、libfabric の `fi_trigger` で用いる `fi_cntr` の最大数 (*max_cts*) や同時投入可能な libfabric の `fi_trigger` 操作の最大数 (*max_triggered_ops*) を確認できる。公開されている BXI のパラメタは textit-max_triggered_ops のみであり、その値は 1024 である [7]。この情報から BXI において貯水槽モデルを実現すると、通信資源であるカウンタの総量は 1024 となる。このようにオフロードのための通信ハードウェア資源であるカウントの数量は非常に限られたものであると考えるべきである。

3. 関連研究

筆者ら [8][9] は、Tofu2 のオフロード機能を用いてブロードキャストアルゴリズムの実装を行い、プログレススレッドを用いた実装と比較しオフロード版の優位性を実証した。しかし、これらは Tofu2 のオフロード API を直接利用したもので汎用性の低い実装となっている。

Ajima ら [2] は、Tofu2 のオフロード機能を利用する例として Bcast や Gather の非同期集団通信のアルゴリズムを報告している。これらはリファレンス実装であり、実用性を考えているものではない。また、永続型集団通信として実装を考慮したものでもない。

また、Hatanaka ら [10] は、Tofu2 のオフロード機能を利用した永続型隣接集団通信の実装を行った。しかし、通信資源をより多く利用する多段の依存関係を持つ永続型集団通信の実装は行われていない。

他にオフロード機能を持つ通信デバイスとしては、Mellanox ConnectX-2 や Bull eXascale Interconnect[3] などが挙げられる。

南里ら [11] は、非同期集団通信の通信隠蔽効果の調査を行った。FX100 のアシスタントコアによる非同期集団通信や Mellanox 社のオフロード機能である SHArP 機能を用いた非同期集団通信の隠蔽効果を調査を行っている。しかし、これらは、スイッチ上での演算を行う reduce が対象であるため、演算をオフロードできない NIC オフロードとは対象が異なる。

Schneider ら [4] は、MPI 非ブロッキング集団通信の汎用実装として使われる libnbc のオフロード対応を提案した。彼らの提案では、MPI send/recv 通信を構成部品として、これらの通信に対するオフロード対応スケジューラを導入することにより、非ブロッキング集団通信を実現している。これは、既存の MPI ブロッキング集団通信の汎用実装が MPI send/recv 関数の上に構築されてきており、これら既存資産を流用するために、send/recv モデルの通信スケジューリングの抽象記述 (cDAG) に書き直すことで非ブロッキング集団通信を実現するのには、一定の合理性がある。集団通信記述の基本部品として send/recv モデルを利用することは、個々の send/recv 通信にオフロード通信のためのハードウェア資源を割り当てることになる。集団通信アルゴリズムが O(Log N) 回の通信を必要とする場合には、O(Log N) 個の send/recv オフロード通信のためのハードウェア資源が必要となり、資源枯渇問題となる可能性が高い。

また、S. Di Girolamo ら [12] は、send/recv モデルを拡張して計算の依存関係を加味する非同期集団通信の実装を行なっている。この実装の通信部分に関しては、Schneider らと同じモデルを用いているため、同様の通信資源が利用される。

4. 設計

前節で述べた通り、Schneider ら [4] の send/recv モデルを利用した永続型集団通信の実装では、各通信ごとにハードウェア通信資源 (カウンタ) の割り当てが行われるため、通信デバイスへのオフロードを考えた場合に通信ステップ数に比例する資源量が必要となる。そこで、集団通信全体として通信発行を通信の受信数より管理することで通信資源を削減する貯水槽モデルの永続型集団通信の設計と実装を提案する。

4.1 Single Source パターン

最も基本的な例として全体にデータを転送する n プロセスの Broadcast を考える。単純な実装として同期通信を実行後、データの発行元であるルートプロセスが n-1 個のプロセスへそれぞれ転送する方法がある。

ここでは、京コンピュータや FX100 で利用される Trinaryx3 アルゴリズムの設計を行う。Trinaryx3 アルゴリズムは分岐通信とパイプライン転送で構成させるので、簡単のため、以下はこれらの通信の貯水槽モデルによる設計について説明する。

4.1.1 分岐通信

分岐通信は自プロセスが受信元プロセスから受信したデータを複数の宛先プロセスへ転送する通信である。Binomial アルゴリズムや Binary アルゴリズムで用いられ、通信のステップ数の削減に利用される。

分岐通信では、複数宛先プロセスへ通信を実行する。このため、貯水槽の水位を分岐数増加させる必要がある。

しかし、DW の利用時では受信元からの通信では水位を表すカウンタ値は 1 しか増加しない。このため、図 4 のように FI_OP_CNTR_ADD で指定される DW 要求をデータ通信の要求の前に挿入する。FI_OP_CNTR_ADD では任意のカウンタの増分値を指定できる。

この要求ではカウンタの増分値を指定する op.cntr.value に通信の分岐数を設定する。このとき、カウンタを増加させる要求の threshold を 1 とする。これにより受信元からメッセージを受信後にカウンタの増加する要求が実行され、分岐通信が発行される。

表 2 に Binary アルゴリズムにおける分岐通信を発行する各プロセスの DW 要求列を示す。Rq は DW 要求の ID、Th に threshold 値、Ot に op_type、Cl に自プロセスのカウンタの増分値、Cr に宛先プロセスのカウンタの増分値を示す。

Binary アルゴリズムは通信の分岐数が 2 である。このため、貯水槽の増分値は 2 となり、貯水槽の増加が終わると水位が 3 となり、分岐通信が 2 個発行される。

4.1.2 パイプライン転送

パイプライン転送はデータをセグメントに分割し、全プ

```
if ( rank == root ) {
  /* 分岐通信のリクエスト */
  for ( i = 1; i < fanout; i++ ) {
    ...;
    dw[i].threshold = n - 1;
    dw[i].op_type = FI_OP_WRITE;
    ...;
  }
 }
else if ( rank != root) {
  /* カウンタを増加するリクエスト生成 */
  ...;
  dw[1].op.cntr.value = fanout;
  dw[1].threshold = 1;
  dw[1].op_type = FI_OP_CNTR_ADD;
  ...;
  /* 分岐通信のリクエスト生成 */
  for ( i = 2; i < fanout + 2; i++ ) {
    ...;
    dw[i].threshold = fanout + 1;
    dw[i].op_type = FI_OP_WRITE;
    ...;
  }
 }
```

図 4 分岐通信の DW 要求列生成コード

表 2 分岐通信を発行するプロセスの DW 要求列（ルート以外）

Rq	Th	Ot	Cl	Cr	DW 要求の内容
0	0	FI_OP_WRITE	-	1	同期通信は即時発行
1	1	FI_OP_CNTR_ADD	2	-	受信後、自カウンタの増加
2	3	FI_OP_WRITE	-	1	分岐通信 1
3	3	FI_OP_WRITE	-	1	分岐通信 2

ロセスで各セグメントの転送処理を並列に実行することで高スループットを実現するものである。

貯水槽モデルで考えると、中間プロセスでは受信元からセグメントが到着したら、水位を 1 とし、水位が 1 となった際にセグメントが発行する。セグメントが発行されたら水位を 0 に戻す、この動作をセグメント数回を繰り返すことでパイプライン転送を実現する。

図 5 にパイプライン転送の DW 要求列生成の疑似コードを示す。

```
if ( rank == root ) {
  for ( i = 0; i < nsegment; i++ ) {
    ...;
    dw[i].threshold = n - 1;
    dw[i].op_type = FI_OP_WRITE;
    ...;
  }
 }
 else if ( rank != root && rank != leaf ){
   for ( i = 0; i < nsegment; i++ ) {
     ...;
     dw[i].threshold = i + 1;
     dw[i].op_type = FI_OP_WRITE;
     ...;
   }
 }
```

図 5 パイプライン転送の DW 要求列生成コード

パイプライン転送の中間プロセスでは、各セグメントの発行する DW 要求の threshold の値をセグメント ID+1 と設定する。中間プロセスが受信元プロセスからの通信を受信した際には、中間プロセスのカウンタが 1 増加する。これにより中間プロセスでは受信したセグメント ID に対応するセグメントが発行させる。

4.2 Multiple Sources パターン

前節では、データの上流と下流の通信相手が固定され、かつ上流は一つである場合、誰からどのメッセージが到着したかは、1 つのカウンタで識別可能であった。この節では、各通信ラウンドで異なる相手と通信する場合について検討する。

4.2.1 貯水槽 2^n 方式

最も基本的な例の一つとして、図 6 に示すような、プロセス数 8 の Butterfly Barrier [13] を考える。この例において、プロセス $P_0 \sim P_7$ はそれぞれ、任意のタイミングで Barrier 操作を開始するので、P_0 に到着するメッセージ $M_{1\to0}$, $M_{2\to0}$ および $M_{4\to0}$ の到着順序は保証されないことに注意が必要である。

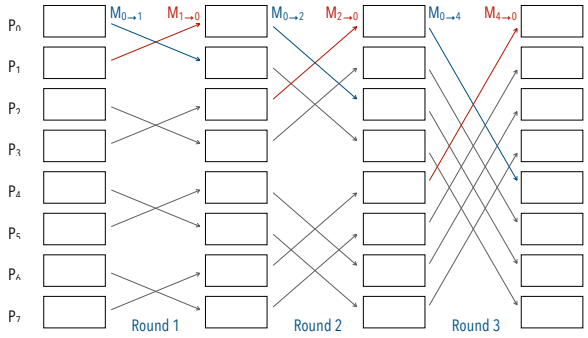

図 6 Butterfly Barrier の概要 (8 プロセス)
記法 $M_{src\to dst}$ は、プロセス P_{src} から P_{dst} へのメッセージである。

よって、プロセス P_0 では、次のような通信スケジューリング・シナリオを実行するスケジューラが必要である。

- Barrier が呼び出されると直ちに、 $M_{0\to1}$ を発行する
- $M_{1\to0}$ を受取った後で、$M_{0\to2}$ を発行する
- $M_{1\to0,2\to0}$ を受取った後で、$M_{0\to4}$ を発行する
- $M_{1\to0,2\to0,4\to0}$ を受取った後で、後始末処理を行う

図 6 で述べた Butterfly Barrier のスケジューリング・シナリオをオフロードするために、節 2.2 で述べた貯水槽モデルで当該シナリオを記述可能である必要がある。まず、P_0 に到着するメッセージ $M_{1\to0,2\to0,3\to0}$ の各受信処理完了イベントそれぞれで、バケツ $Bucket_{1,2,3}$ から貯水槽への注水が生じ、各注水は任意のタイミング (順序) で発生するものとする。このとき、次のような条件を満たすように、バケツの水の量 (ならびに水位スイッチの設定水位) を調整したい。

- バケツ $Bucket_i$ が注水済みであるとき、水位スイッチ $Swtich_i$ がオンになる
- 但し、$1 \le j \le i$ を満たすすべての $Bucket_j$ が注水済みでないと、$Switch_i$ はオンにならないし、$j > i$ であるいかなる $Bucket_j$ も、$Switch_i$ の切替に影響を与えない

例えば、$Bucket_1$ を注ぐ前に、先に $Bucket_2$ と $Bucket_3$

が注水済みであったとしても、$Switch_1$ はもちろん、$Switch_2$ も $Switch_3$ もオンにならないように調整したい。この場合、図 7 ように 2 の冪乗の性質を使えば、上記の水位スイッチの発報条件を満たす、バケツの水の量と水位スイッチの設定水位を選ぶことが可能である。これを「貯水槽 2^n 方式」と呼ぶ。

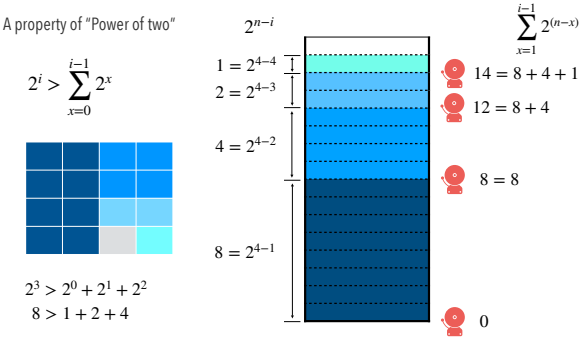

図 7 2 の冪乗と貯水槽 2^n 方式
図中の n を必要な Butterfly ラウンド数、i を 1 から始まる 各ラウンドとする。

4.2.2 DW (Deferred Work) 要求の拡張

貯水槽 (Water Tank) モデルに従い、Butterfly Barrier 集団通信アルゴリズムに対し libfabric の DW (Deferred Work) 要求の列として記述を試みる。

しかしながら、libfabric の DW 要求には、リモートのカウンタに対し任意の値を加算させる操作が提供されていないため、下位のハードウェア違いを隠蔽することができない。このため、図 8 に示すように、既存の DW 要求を拡張して、抽象記述が可能な新しい操作を導入する。まず、フィールド *op_type* に対し新タイプ `FI_OP_REMOTE_CNTR_ADD` を導入する。次に、フィールド *op* に対し対応する操作のパラメタを記述する `fi_op_remote_cntr` 構造体及び `remote_cntr` を追加する。また、新操作 `fi_op_remote_cntr` のフィールドの意味については表 3 に示す。

```
struct fi_op_remote_cntr {
  fi_attr_t addr;
  uint64_t cntr_key;
  uint64_t value;
};

struct fi_deferred_work { ...
  uint64_t threshold;
  struct fid_cntr * triggering_cntr; ...
  enum fi_tringer_op op_type;
  union { ...
    struct fi_op_remote_cntr * remote_cntr;
  } op;
};
```

図 8 DW (Deferred Work) 要求の拡張

4.2.3 Butterfly Barrier のオフローディング

Barrier はユーザデータの転送を含まない特殊な集団通信である。図 6 で示した Butterfly Barrier アルゴリズムでは、各ラウンド r の開始時点で、前段までの pair-wise の通信を通して 2^{r-1} のメンバが集団通信を開始済みであることが保証される。

表 3 `FI_OP_REMOTE_CNTR_ADD` 操作のフィールド

略号	dw フィールド名	備考
Pq	`.op.remote_cntr->addr`	対象の貯水槽倉庫の住所
Ci	`.op.remote_cntr->cntr_key`	対象の貯水槽の識別子
Iv	`.op.remote_cntr->value`	対象の貯水槽への *Bucket* 注水量

この Butterfly Barrier を、前節 4.2.2 で示した libfabric の拡張 DW (Deferred Work) 要求の列として記述すると、表 4 のようになる。プロセス $P_0 \sim P_7$ はそれぞれ、Butterfly Barrier 用カウンタ C_{bb} を有する。これは本モデルでの貯水槽に相当し、その初期水位は 0 とする。

表 4 Butterfly Barrier の DW 要求列 (図 6 の例での P_0)
略号 `Th,Ot,Iv,Pq,Ci` については、表 1 および 3 を参照せよ。

r	Th	Ot	Iv	Pq	Ci
1	0	FI_OP_REMOTE_CNTR_ADD	4	1	C_{bb}
2	4	FI_OP_REMOTE_CNTR_ADD	2	2	C_{bb}
3	6	FI_OP_REMOTE_CNTR_ADD	1	4	C_{bb}
C	7	FI_OP_REMOTE_CNTR_ADD	-7	0	C_{bb}

総数 n ラウンドの各ラウンド r で、Th は $Switch_r$ の設定水位に相当し、$\sum_{x=1}^{r-1} 2^{(n-x)}$ に設定される。Iv は $Bucket_r$ の水の量に相当し、$2^{(n-r)}$ に設定される。プロセス P_p での各ラウンド r での通信相手プロセス Pq は `q = p ^ (1 << (r-1))` に設定される。

初回ラウンドでは、自身が集団通信を開始済みであることを通知するために、直ちに最初の `FI_OP_REMOTE_CNTR_ADD` 操作が実行されなければならない。このために、C_{bb} の初期値 0 に対し $\mathtt{Th} = 0$ に設定することで、$\mathtt{Th} \geq C_{bb}$ の発報条件を満たし、直ちに実行可能状態になる。完了ラウンド C $(= n + 1)$ では C_{bb} の値は $\sum_{x=1}^{n} 2^{(n-x)}$ になっており、`FI_OP_CNTR_SUB` 操作は定義されないので、-7 を加算し、自身のカウンタ C_{bb} を巡回 (wrap around) させて 0 に戻す。

プロセス P_0 では、P_1 からの `FI_OP_REMOTE_CNTR_ADD` によって C_{bb} は 4 つ増加する。同様に、P_2 によって 2 つ増加し、P_4 によって 1 つ増加する。

4.2.4 Butterfly Allgather のオフローディング

Barrier と異なりユーザデータの転送を含む場合には、Butterfly 通信の各ラウンドの pair-wise 通信で、次の 2 つ条件を満たさない限り、データ転送を開始できない。つまり、(1) 送信側で送信すべきデータがすべて揃ったこと、かつ (2) 受信側で受信バッファの準備が整ったこと、の 2 つである。

こうしたデータ依存関係を拡張 DW 要求列として記述する例として、同じ Butterfly ネットワークの Butterfly Allgather (Recursive Doubling) を考える。この場合、各 Butterfly 通信ラウンドでは pair-wise のデータ交換が発生

する。典型的な RDMA ベースの Receiver-Initiated Rendezvous プロトコルで、各ラウンドの交換を整理すると、図 9 のようになる。

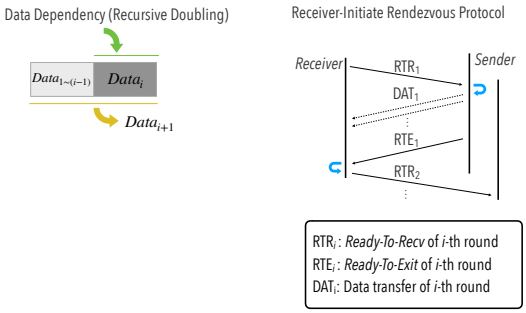

図 9 データ依存関係の記述と Allgather への展開
記法 $M_{src \to dst}$ は、プロセス P_{src} から P_{dst} へのメッセージである。

データ交換の場合、送信側/受信側を兼務するので、まず前ラウンドまでのデータが揃った時点で、相手に準備が整ったことを通知するために RTR (Ready-To-Recv) を発行する。次に相手からの RTR を受信した時点で、実際のデータ転送 DAT (RDMA-WRITE) を発行する。すべての DAT 発行後に RTE (Ready-To-Exit) を発行する。相手から RTE 受信した時点で、当該ラウンドのデータは揃っているので、次のラウンドに進む。

表 5 Butterfly Allgather の DW 要求列 (図 6 の例での P_0)
Mn は便宜上のメッセージ名。r は論理ラウンド。R は実ラウンド。Ot 列の ot_1 は FI_OP_REMOTE_CNTR_ADD 、ot_2 は FI_OP_WRITE である。略号 Th,Ot,Iv,Pq については、表 3 を参照せよ。

Mn	r	R	Th	Ot	Iv	Pq
RTR_1	1	1	0	ot_1	32	1
DAT_1	1	2	32	ot_2	0	1
RTE_1	1	2	32	ot_1	16	1
RTR_2	2	3	48	ot_1	8	2
DAT_2	2	4	56	ot_2	0	2
RTE_2	2	4	56	ot_1	4	2
RTR_3	3	5	60	ot_1	2	4
DAT_3	3	6	62	ot_2	0	4
RTE_3	3	6	62	ot_1	1	4
FIN_C	C	7	63	ot_1	-63	0

このように、データ交換パターンでは、各 Butterfly 通信ラウンド (図中 r) に対し 2 つのチェック・ポイントが必要なので、実ラウンド (R) は Butterfly 通信ラウンドの二倍の段数が必要になる。つまり、貯水槽モデルのスイッチが二倍に増える。

また、RTR は、データ受信側のバッファの準備が整ったことを通知するための制御メッセージであったので、データの配置上、各ラウンドの転送データが全ラウンドを通して独立したメモリ領域を保証できる (破壊されない) なら、RTR を集団通信に入った時点 (Th = 0) で全員に通知することもできる。

4.2.5 貯水槽 2^n 方式の展開

前節の Butterfly Allgather の方式では、RTE によるリモートカウンタの実行の前に、先行する DAT の実行結果がリモートメモリに反映されている必要がある。例えば、発行側で先行する DAT の到達確認を待って RTE をスケジュールしたり、応答側で先行する DAT のメモリアクセスの完了まで RTE の実行を遅らせる等のさまざまなハードウェア機構が提供されているが、ハードウェアの実装に依存するので、ここでは詳しく述べない。

このように、貯水槽の制御から、DAT を切り離すことで、butterfly bcast (段数が増えるが) や butterfly all-to-all 等異なる集団通信であっても、同じ通信パターン (各ラウンドの通信相手すべてが同じ) なら同じ貯水槽を共用する可能性が広がる。

5. 評価

5.1 Broadcast のリソース量

5.1.1 カウンタ量

京コンピュータで用いられている Trinaryx3 アルゴリズム [14] を用いて永続型集団通信 Broadcast の通信ハードウェア資源量について考える。

提案した貯水槽モデルの実装では、受信元にのみに貯水槽が必要となる。Trinaryx3 アルゴリズムは 3 分木の分岐通信とパイプライン通信を組み合わした集団通信アルゴリズムである。この 3 分木の分岐通信では親となる受信元プロセスは 1 個となるので、自プロセスにはカウンタが 1 個だけ必要となる。Trinaryx3 アルゴリズムでは、3 種類のパイプラインパスを生成するため、合計でカウンタが 3 個で必要となる。

一方、send/recv モデルの実装では、汎用的な実装になっており、基本となる一対一通信においては受信用カウンタは 2 個、送信用カウンタは 1 個必要となる。Trinayx3 アルゴリズムでは、3 種類のパイプラインパスを生成するので受信用カウンタが 6 個必要となる。送信側は、各パプラインパスで 1〜3 個の分岐数の分岐通信を行うが各プロセスは総計で 3 個以上の通信を発行することはないので送信用のカウンタは 3 個となる。これらを合わせると 9 個のカウンタが必要となる。

send/recv モデルの実装では、提案した貯水槽モデルには必要がない通信の到着確認 (tag-maching) のための通信資源が必要となる。通信デバイスにオフロードする場合は、これらはハードウェア資源等から最大量の制限がある。

Trinaryx3 アルゴリズムは各プロセスでパイプライン転送を行う。パイプライン転送では 3 個の上流プロセスから通信を受信し、確認を行う。このため、各プロセスのパイプライン転送ではセグメント数 × 3 個の通信が受信される。

パイプライン転送の適切なセグメント分割数は、レイテンシ、スループット、通信間のオーバヘッドの影響を受ける。今回は、京コンピュータ、および FX100 の 1Tofu 単位を用いてセグメント分割数を実測で求めた [9]。これから、send/recv モデルでパイプライン転送を実現しようとすると、128MB データ転送時に、京コンピュータ、FX100 のそれぞれで 2049 回、129 回の send/recv、すなわち、ハードウェア通信資源であるカウンタ数がそれぞれ 2049 個、129 個必要となる。

節 2.3 で述べたとおり、BXI の場合、ハードウェア通信資源の数は高々 1024 であり、セグメント分割数はこの数で制限される。このように、send/recv モデルにおけるパイプライン転送処理実現では、ハードウェア通信資源制限により最適なセグメント分割数を選択できない可能性がある。

5.2 Butterfly Allgather のリソース量

節 4.2.4 で述べたように、Butterfly Allgather は、そのコミュニケータに属するメンバの数にしたがって、段数が増大する。表 6 は、京 および 100 万ノード規模の計算機での、オフロード用通信資源である貯水槽 (カウンタ) の要求量を示す。但し、ノード内 (intra-node) 通信に関しては、異なるアルゴリズムで実現することができるので、ここではノード間 (inter-node) 通信についての通信資源量を求める。

表 6 貯水槽 2^n と pre-matched 方式のカウンタ数比較

N_n	$floor(log_2(N_n))+2$	`WaterTank` 2^n	`pre-matched`
82,944	$18=16+2$	$1(R=36)$	$54=18\times3$
1,000,000	$21=19+2$	$1(R=42)$	$63=21\times3$

Butterfly 通信では、全体ノード数が二のべき乗でない場合、一般に前後合わせて +2 ラウンド増える。貯水槽 2^n 方式の場合、さらにデータ転送を含むなら、各ラウンドで 2 つのチェックポイントが必要となるが、カウンタは一つでよい。しかし、カウンタは 32 ビットでは表現できないので、64 ビットのカウンタが望ましい。

一方で、Schneider ら [4] の永続型に適した pre-matched 方式は、send/recv モデルであるため、各ラウンドで 3 基のカウンタ (送信側 1+受信側 2) が必要になり、100 万ノードで 63 基のカウンタが必要である。また、永続集団通信の init 関数から free 関数までの生存時間が重なる場合、それぞれの集団通信に対して、別セットの資源を割り当てる必要がある。たとえ、同じコミュニケータ上の複数の allgather でも、それぞれ別のセットを準備する必要がある。

6. おわりに

本稿では、オフロード API である libfabric の `fi_trigger` を用いて、オフロード永続型集団通信の設計を行った。また、その際のハードウェア資源の消費量の評価を行い、要求資源が十分に少ないことを示した。

今回は、比較的に簡易に実装が可能な Broadcast のような Single Source パターンの集団通信だけでなく、永続型集団通信の実装において複数ソースからの通信がある allgather や barrier などの Multipile Soruces パターンの集団通信において MPI send/recv のセマンティクスの実現に通信資源を奪われることなく、より単純でハードウェア寄りの `fi_trigger` と呼ばれるオフロード機能を用いて集団通信オフローディングの実現方式を提案した。これらにより、今まであまり考慮されていなかった通信資源を抑えた永続型の集団通信の汎用実装が可能となったと考える。

今後の課題としては、設計した永続型集団通信の性能評価や他の永続型集団通信での資源量および性能に関する評価の実施があげられる。

謝辞 本論文の一部は、「特定先端運営費等補助金/ 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る。

参考文献

[1] Message Passing Interface Forum. MPI: 2018 Draft Standard. `https://www.mpi-forum.org/docs/drafts/mpi-2018-draft-report.pdf`.

[2] Y. Ajima, T. Inoue, S. Hiramoto, S. Ando, M. Maeda, T. Yoshikawa, K. Hosoe, and T. Shimizu. Tofu interconnect 2. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pp. 57–62, aug 2014.

[3] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. The bxi interconnect architecture. In *IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI'15)*, pp. 18–25, aug 2015.

[4] Timo Schneider, Torsten Hoefler, Ryan E. Grant, Brian W. Barrett, and Ron Brightwell. Protocols for Fully Offloaded Collective Operations on Accelerated Network Adaptors. In *2013 42nd International Conference on Parallel Processing*, Sep 2013.

[5] Libfabric programmer's mannual. `https://ofiwg.github.io/libfabric/`.

[6] Portals 4 specification. `http://www.cs.sandia.gov/Portals/portals4-spec.html`.

[7] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. The bxi interconnect architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 18–25, Aug 2015.

[8] Y. Morie, M. Hatanaka, M. Takagi, A Hori, and Y. Ishikawa. Prototyping of offloaded persistent broadcast on tofu2 interconnect. In *SC17*, nov 2017.

[9] 森江善之, 畑中正行, 高木将通, 堀敦史, 石川裕. Fx100 における永続型集団通信関数のプロトタイプ実装と評価. 情報処理学会研究会報告ハイパフォーマンスコンピューティング（HPC), 第 2018-HPC-163 巻, feb 2018.

[10] M. Hatanaka, M. Takagi, A Hori, and Y. Ishikawa. Offloaded mpi persistent collectives using persistent generalized request interface. In *EuroMPI/USA 2017*, sep 2017.

[11] 南里豪志, 大島聡史, 小野謙二. 非ブロッキング集団通信の通信隠蔽効果に関する調査. 情報処理学会研究会報告ハイパフォーマンスコンピューティング (HPC), 第 2017-HPC-162 巻, pp. 1–11, dec 2017.
[12] S. Di Girolamo, P. Jolivet, K. D. Underwood, and T. Hoefler. Exploiting offload-enabled network interfaces. *IEEE Micro*, Vol. 36, No. 4, pp. 6–17, aug 2016.
[13] Eugene D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, Vol. 15, No. 4, pp. 295–307, Aug 1986.
[14] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa. 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, may 2013.