FUJITSU

FUJITSU Software
Technical Computing Suite V4.0L20

Development Studio
uTofu使用手引書

J2UL-2567-01Z0(03)
2021年7月

まえがき

本書の目的

本書は、Tofuインターコネクトを使用して通信を行うためのプログラミングインターフェースであるuTofuの使用方法を説明しています。

本書では、富士通製CPU A64FXを搭載したシステムを「本システム」と呼びます。

本書の読者

本書は、Tofuインターコネクトを使用して通信を行うプログラムを開発する人を対象に記述しています。本書を読むにあたっては、Cプログラムならびに Linuxのコマンド、ファイル操作、およびシェルプログラミングの基本的な知識が必要です。

本書の構成

本書は、以下の構成になっています。

第1章 TofuインターコネクトとuTofuの概要

TofuインターコネクトとuTofuの概要を説明します。

第2章 uTofuの通信モデル

uTofuの通信モデルについて説明します。

第3章 uTofuインターフェース仕様

uTofuのインターフェース仕様について説明します。

第4章 uTofuの使用法

uTofuの使用法について説明します。

第5章 uTofuの使用例

uTofuを使用したいくつかの例を示します。

第6章 システムの情報

システム固有の情報について説明します。

第7章 エラーメッセージ

uTofuの実装が出力するエラーメッセージについて説明します。

用語集

用語について説明します。

本書の位置付け

本書は、以下のマニュアルと関係があります。これらのマニュアルについても併せてお読みください。

- MPI使用手引書
- C言語使用手引書
- C++言語使用手引書
- Fortran使用手引書

上記以外に、関連ソフトウェアであるジョブ運用ソフトウェアのマニュアルも必要に応じて参照してください。

また、本書にはMPIに関する記述が一部含まれます。MPIについては、以下の規格書を参照してください。

MPI: A Message-Passing Interface Standard Version 3.1 Message Passing Interface Forum June 4, 2015

MPIに関する情報は、https://www.mpi-forum.org/ から入手することができます。

単位の表現

本書では、単位を表現する際の接頭語を以下のとおり使い分けています。

接頭語	値	接頭語	値
k (kilo)	10^3	Ki (kibi)	2^{10}
M (mega)	10^6	Mi (mebi)	2^{20}
G (giga)	10^9	Gi (gibi)	2^{30}

構文表記記号

構文表記記号とは、構文を記述するうえで特別な意味で定められた記号であり、以下のものがあります。

記号名	記号	説明
選択記号	{ }	この記号で囲まれた項目の中から、どれか1つを選択することを表します。
	|	この記号を区切りとして、複数の項目を列挙することを表します。
省略可能記号	[ ]	この記号で囲まれた項目を省略してよいことを表します。また、この記号は選択記号“{ }”の意味を含みます。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

商標

- Linux(R)は米国及びその他の国におけるLinus Torvaldsの登録商標です。
- そのほか、本マニュアルに記載されている会社名および製品名は、それぞれ各社の商標または登録商標です。

出版年月および版数

版数	マニュアルコード
2021年 7月 第1.3版	J2UL-2567-01Z0(03)
2020年 9月 第1.2版	J2UL-2567-01Z0(02)
2020年 6月 第1.1版	J2UL-2567-01Z0(01)
2020年 2月 初版	J2UL-2567-01Z0(00)

著作権表示

変更履歴

変更内容	変更箇所	版数
以下の環境変数を追加しました。 ・ UTOFU_SWAP_PROTECT	4.3.2.5	第1.3版
通信集中時の説明を加筆しました。	4.1.4	第1.2版
TNIあたりの使用可能CQ数を変更しました。	6.1.2	

変更内容	変更箇所	版数
仮想化された資源の量について説明を追加しました。		
「エラーメッセージ」を追加しました。	第7章	
「Tofuインターコネクトのリンクダウン時に通信経路を変更する設定がされている場合の動作」を追加しました。	6.1.4.2	第1.1版
説明文を見直しました。	―	

目　次

第1章 Tofuインターコネクトとu Tofuの概要

この章では、Tofuインターコネクトとu Tofuの概要を説明します。

u Tofuは、Tofuインターコネクトで通信を行うプログラムを作成するためのプログラミングインターフェースです。

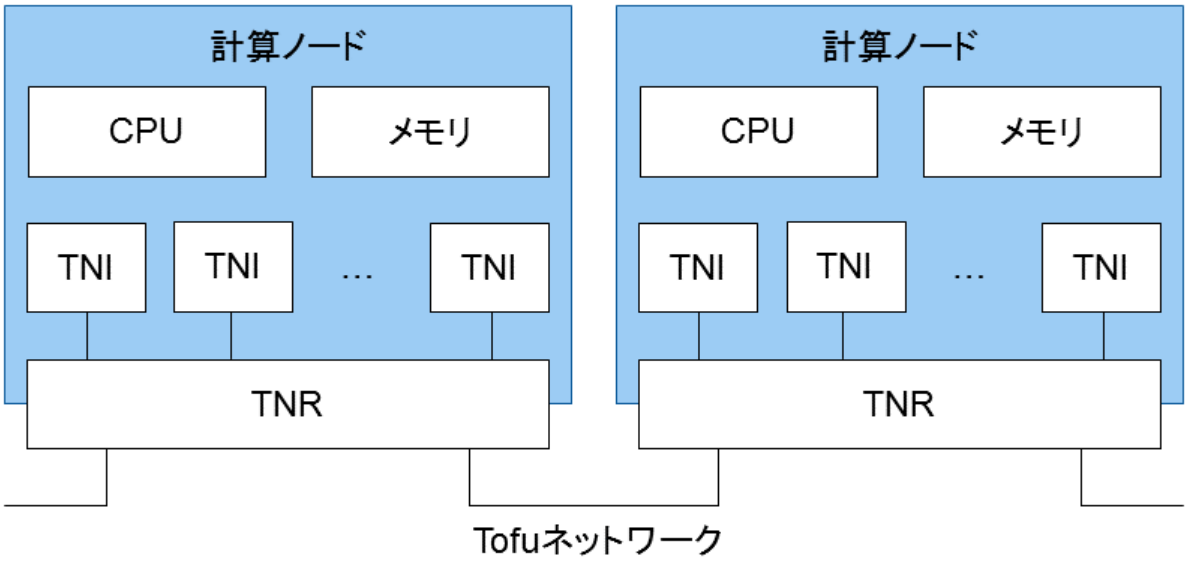
1.1 Tofuインターコネクトの概要

1.1.1 アーキテクチャ

Tofuインターコネクトは、Tofuネットワーク、Tofuネットワークインターフェース(TNI)、およびTofuネットワークルータ(TNR)から構成される、並列計算機システム向けインターコネクトです。本書では、ハイエンドテクニカルコンピューティングサーバMP10システムとスーパーコンピュータPRIMEHPC FX10システムで使用されているTofuインターコネクト、スーパーコンピュータPRIMEHPC FX100システムで使用されているTofuインターコネクト2、および本システムで使用されているTofuインターコネクトDを総称して、Tofuインターコネクトと呼びます。

これらの並列計算機システムは複数の計算ノードから構成され、1つの計算ノードは1つ以上のTNIと1つのTNRを持ちます。TNRを相互接続することにより、Tofuネットワークが構成されます。TNIはソフトウェアからの指示にしたがってパケットを送受信します。TNRとTofuネットワークは送信元TNIから宛先TNIまでパケットを転送します。1つの計算ノード上で2つ以上のTNIが同時に処理を行うことにより、高スループットの通信が可能になっています。計算ノードの構成を下図に示します。

図1.1 計算ノードの構成

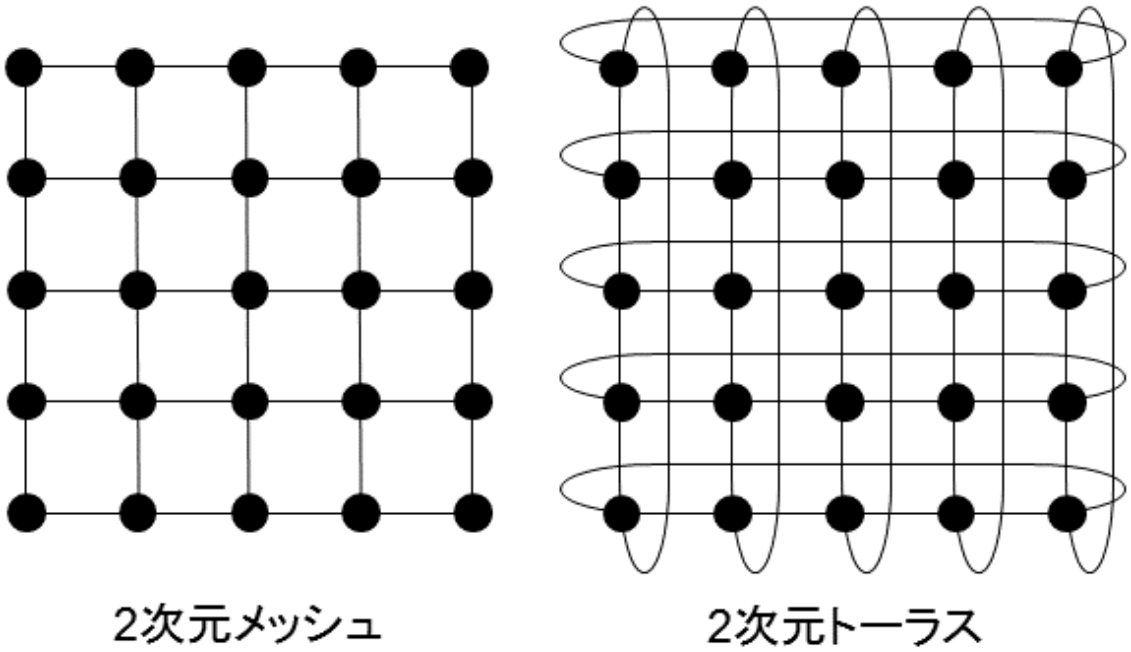


1.1.2 ネットワーク

Tofuインターコネクトは、複数の計算ノードをTofuネットワークで接続します。

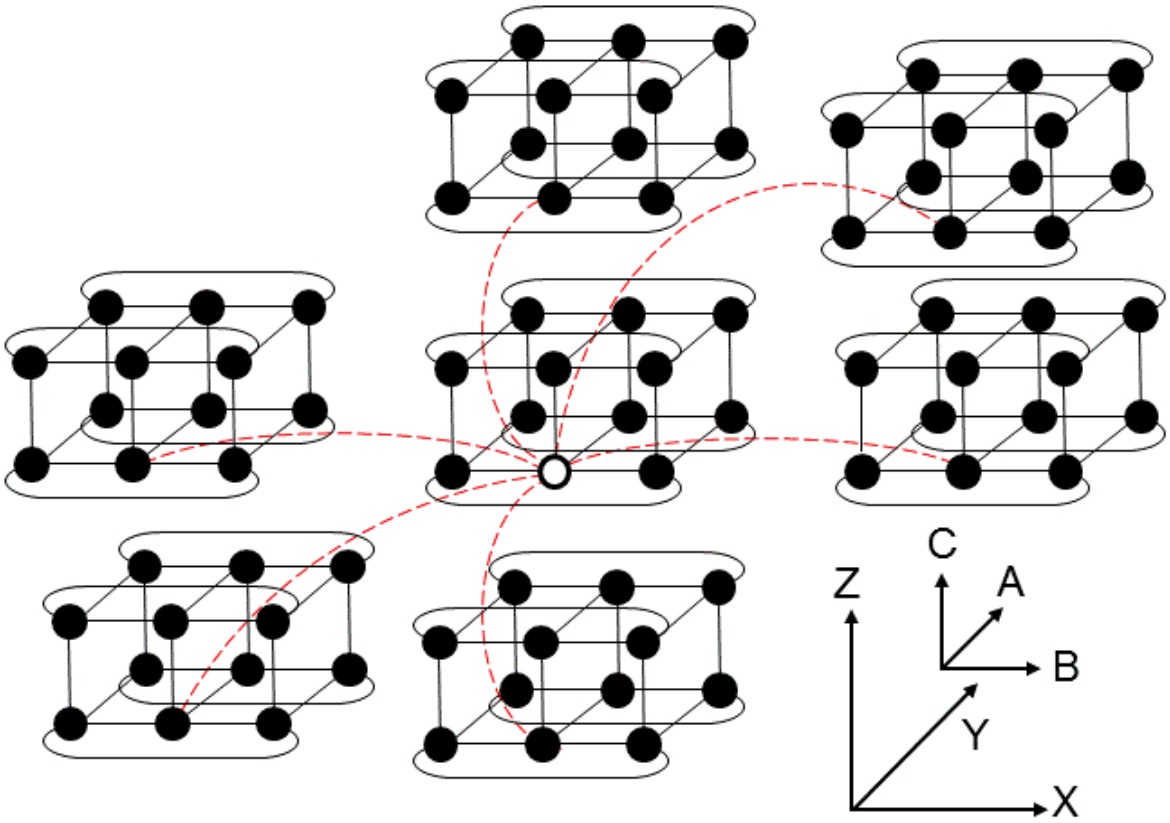
Tofuネットワークは6次元のメッシュ/トーラスのトポロジーを持ちます。メッシュとは、座標軸の端点が相互に接続されていないトポロジーを指します。トーラスとは、座標軸の端点が相互に接続されているトポロジーを指します。2次元でのメッシュとトーラスの例を下図に示します。

図1.2 2次元でのメッシュとトーラスの例



Tofuネットワークの6次元の座標軸にはX、Y、Z、A、B、Cの文字が与えられています。A、B、Cの3次元は、長さが2、3、2固定であり、B軸はトーラスで接続されています。X、Y、Zの3次元は、システムによって、長さが異なり、メッシュで接続されているかトーラスで接続されているかも異なります。そのため、1つの計算ノードのTNRから、A、C方向にはそれぞれ1本のTofuリンク、B方向には2本のTofuリンク、メッシュの端点ではない計算ノードのX、Y、Z方向にはそれぞれ2本のリンク、メッシュの端点となる計算ノードのX、Y、Z方向にはそれぞれ1本のリンクが伸び、合計で最大10本のTofuリンクが伸びています。この6次元の接続を下図に示します。この図では、直方体で表されているA、B、C軸はすべての接続が実線で描かれていますが、X、Y、Z軸は中央の1つの計算ノードに着目しその計算ノードから伸びる接続だけが破線で描かれています。

図1.3 Tofuネットワークの6次元の接続



各計算ノードは、この6次元の座標X、Y、Z、A、B、Cで識別できます。この座標を計算ノード座標と呼びます。

6次元のメッシュ/トーラス内で隣接していない(Tofuリンクで直接接続されていない)計算ノードの間の通信は、その間の計算ノードのTNRによって中継されます。

1.1.3 通信方式

Tofuインターコネクトでは、ユーザー空間からはワンサイド通信とバリア通信という2つの通信方式で通信を行うことができます。

1.1.3.1 ワンサイド通信

ワンサイド通信は、ある計算ノード(ローカル計算ノード)が別の計算ノード(リモート計算ノード)の主記憶をCPUを介さずに参照・更新する通信方式であり、Put、Get、およびARMW(Atomic Read Modify Write)の3種類の通信機能を持ちます。この通信方式は一般にRDMA(remote direct memory access)と呼ばれます。

Putは、ローカル計算ノードの主記憶上の連続データを読み込み、そのデータをリモート計算ノードの主記憶に書き込む通信機能です。Getは、リモート計算ノードの主記憶上の連続データを読み込み、そのデータをローカル計算ノードの主記憶に書き込む通信機能です。ARMWは、リモート計算ノードの主記憶上の4バイトまたは8バイトのデータに対して読込み、演算、および書込みを不可分に行う通信機能です。

ローカル計算ノードとリモート計算ノードの読込み・書込み対象のメモリ領域や、ARMWの演算種別・オペランドは、すべてローカル計算ノードのソフトウェアが指定します。

ワンサイド通信はユーザー空間のプロセスからカーネルを経由せずに実行できます。

1.1.3.2 バリア通信

バリア通信は、複数の計算ノード間でバリア同期を行う通信機能です。バリア通信に参加するプロセスは、あらかじめバリア通信の設定をしておきます。各プロセスは、バリア通信を開始し、その完了を待ちます。バリア通信に参加しているすべてのプロセスがバリア通信を開始すると、各プロセスでバリア通信が完了します。バリア同期と同時にリダクション演算を実行することもできます。

バリア通信はMPIのMPI_BARRIERルーチンやMPI_ALLREDUCEルーチンなどの実装に使うことができます。

バリア通信はユーザー空間のプロセスからカーネルを経由せずに実行できます。

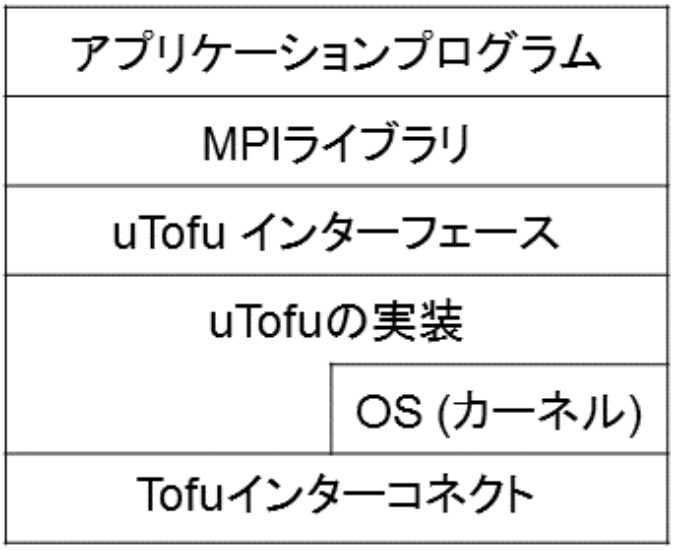
1.2 uTofuの概要

uTofuは、ユーザー空間のソフトウェアがTofuインターコネクトを使用して通信を行うための、低レベルなアプリケーションプログラミングインターフェース(API)です。uTofuはTofuインターコネクトのワンサイド通信とバリア通信をサポートします。uTofuのインターフェースはC言語の関数の形式で提供されます。uTofuの実装はライブラリの形式で提供されます。本書ではこの実装をuTofu実装と呼びます。

uTofuは、アプリケーションプログラムがプロセス間で通信を行うためのライブラリなど(例えばMPIライブラリ)から使用されることを想定して設計されています。

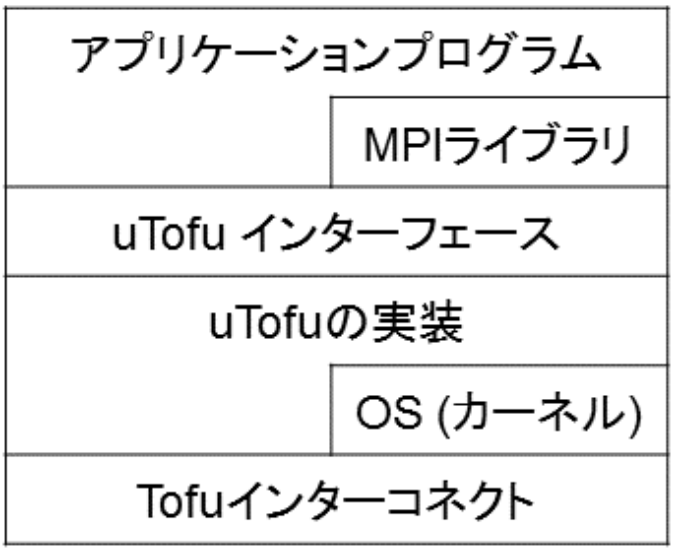
MPIライブラリから使用される場合の例を、“図1.4 MPIライブラリから使用される場合のソフトウェアスタックの例”に示します。

図1.4 MPIライブラリから使用される場合のソフトウェアスタックの例



しかし、“図1.5 アプリケーションプログラムから直接使用される場合のソフトウェアスタックの例”に示すように、MPIで書かれた通信をチューニングするためにアプリケーションプログラムから直接使用することも可能です。

図1.5 アプリケーションプログラムから直接使用される場合のソフトウェアスタックの例



また、同一プロセス内のスレッド間や同一スレッド内で通信(メモリアクセス)を行うために使用することもできます。本書では、uTofuを使用するライブラリやアプリケーションプログラムを、uTofuプログラムと呼びます。また、uTofuプログラムを実行した際のプロセスを、uTofuプロセスと呼びます。

uTofuは、MPIのランクやグループのような複数のプロセスを識別・管理する機能を持ちません。そのような機能が必要な場合は、MPIのような機構と組み合わせるか、独自に作成する必要があります。

uTofuでは、通信の開始と完了確認を行う関数が分離されています。通信の開始を行う関数は、すぐに通信を開始できない場合に、それを示す復帰値を返却します。通信の完了確認を行う関数は、通信が完了したか完了していないかを示す復帰値を返却します。そのため、すべての関数は、通信相手のプロセス・スレッドの状態にかかわらず、有限の時間で復帰します。また、確実に通信を行うには、関数の復帰値を確認する必要があります。

第2章 uTofuの通信モデル

この章では、uTofuの通信モデルについて説明します。

2.1 実行単位

uTofuの実行単位はOS上のユーザー空間のプロセスであり、その実行単位ごとに通信コンテキストを持ちます。これは例えば以下を意味します。

- あるプロセスが作成・確保した通信資源をほかのプロセスが直接使用できない。
- あるスレッドが開始した通信の完了を同じプロセスのほかのスレッドが確認できる。

uTofuは1つのプロセス内で通信コンテキストを分離する機能(VCQ、VBG)を持ちます。この機能を使えば、1つのプロセス内で、複数のソフトウェアコンポーネント(MPIライブラリや並列言語ランタイム環境など)または複数のスレッドが、論理的に独立して通信を行うことができます。

2.2 ワンサイド通信

ワンサイド通信では、TNIが計算ノードの主記憶を直接参照・更新しパケットを送受信することにより、2つの計算ノード間で通信を行います。

2.2.1 VCQ(Virtual Control Queue)

TNIはCQ(control queue)を通してソフトウェアから操作できます。TNIあたりのCQの数は有限であり、Tofuインターコネクトの種類によって異なります。そのため、uTofuでは、ハードウェア資源であるCQをVCQ(virtual control queue)として仮想化・抽象化しています。uTofuによるすべてのワンサイド通信は、このVCQを通して行われます。

あるVCQを通して行われる通信は、ほかのVCQを通して行われる通信の論理的な動作に影響を及ぼしません。例えば、あるVCQを通して開始された通信の完了は、そのVCQだけに通知されます。

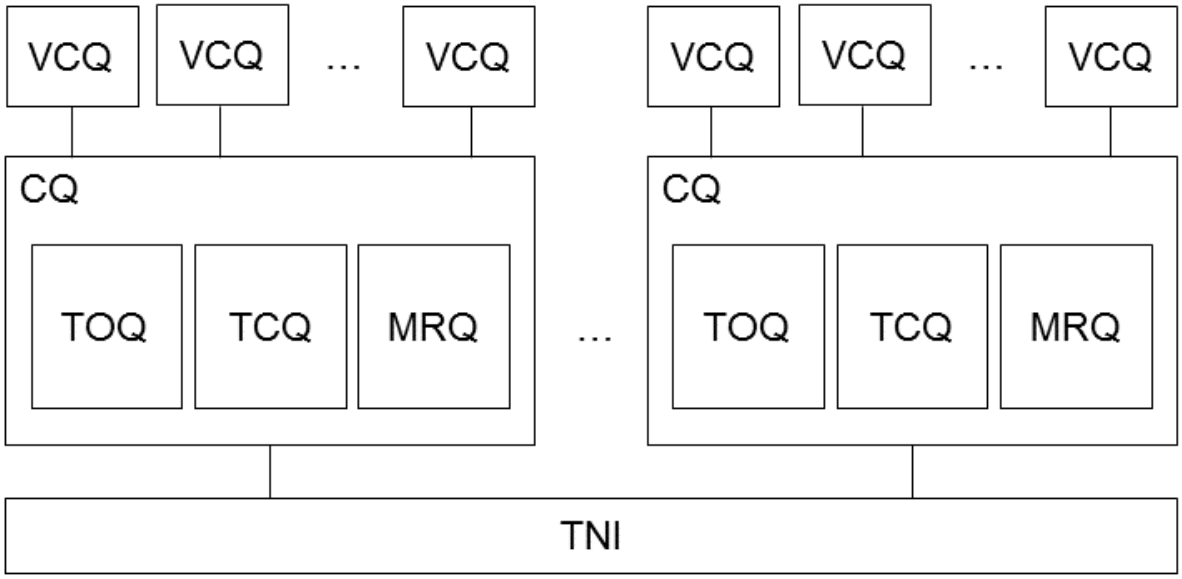
あるプロセスがVCQを作成すると、そのプロセス内で有効なVCQハンドルが得られます。uTofuのすべてのワンサイド通信の関数は、VCQハンドルを引数に持ちます。また、それぞれのVCQにはVCQ IDが与えられます。VCQ IDは、VCQを特定する64ビット符号なし整数であり、Tofuネットワーク内のすべてのVCQ間で一意の値を持ちます。

Tofuインターコネクトでは、2つのCQ間でワンサイド通信を行います。片方のCQはソフトウェアから通信の指示を直接与えられたCQ(ローカル)であり、他方のCQは通信の対象となるCQ(リモート)です。uTofuでワンサイド通信を行う場合は、uTofuの関数の引数に、ローカルのVCQハンドルとリモートのVCQ IDを与えます。そのため、通信を開始する前に、リモートのVCQ IDを知っておく必要があります。

CQおよびVCQは、役割の異なるTOQ、TCQ、およびMRQの3つのキューから構成されます。ソフトウェアがこれら3つのキューに対してディスクリプタと呼ぶデータの読み書きをすることで、通信の指示と完了確認を行います。

TNI・CQ・VCQの構成を下図に示します。

図2.1 TNI・CQ・VCQの構成



2.2.1.1 TOQ(Transmit Order Queue)

TOQ(transmit order queue)は送信指示キューです。ソフトウェアがTNIに対して通信の指示を出すために使用します。ソフトウェアがディスクリプタを書き込み、TNIがそのディスクリプタを読み込みます。このディスクリプタをTOQディスクリプタと呼びます。

ワンサイド通信には複数の種類の通信があり、通信の種類に応じてTOQディスクリプタは細分化されます。通信の種類とディスクリプタ名の対応を下表に示します。細分化されたTOQディスクリプタを、Putディスクリプタなどと呼びます。なお、NOPは通信ではありませんが、ディスクリプタが存在するため、便宜上、この表に含めています。

表2.1 通信の種類とディスクリプタの対応

通信の種類	ディスクリプタ名
Put	Put
	Put Piggyback
Get	Get
ARMW	ARMW
NOP	NOP

2.2.1.2 TCQ(Transmit Complete Queue)

TCQ(transmit complete queue)は送信完了キューです。TNIがソフトウェアに対してデータ送出の完了を通知するために使用します。TNIがディスクリプタを書き込み、ソフトウェアがそのディスクリプタを読み込みます。このディスクリプタをTCQディスクリプタと呼びます。

2.2.1.3 MRQ(Message Receive Queue)

MRQ(message receive queue)はメッセージ受信キューです。TNIがソフトウェアに対して通信の完了を通知するために使用します。TNIがディスクリプタを書き込み、ソフトウェアがそのディスクリプタを読み込みます。このディスクリプタをMRQディスクリプタと呼びます。MRQディスクリプタは、ローカル通知とリモート通知の2種類に分類されます。

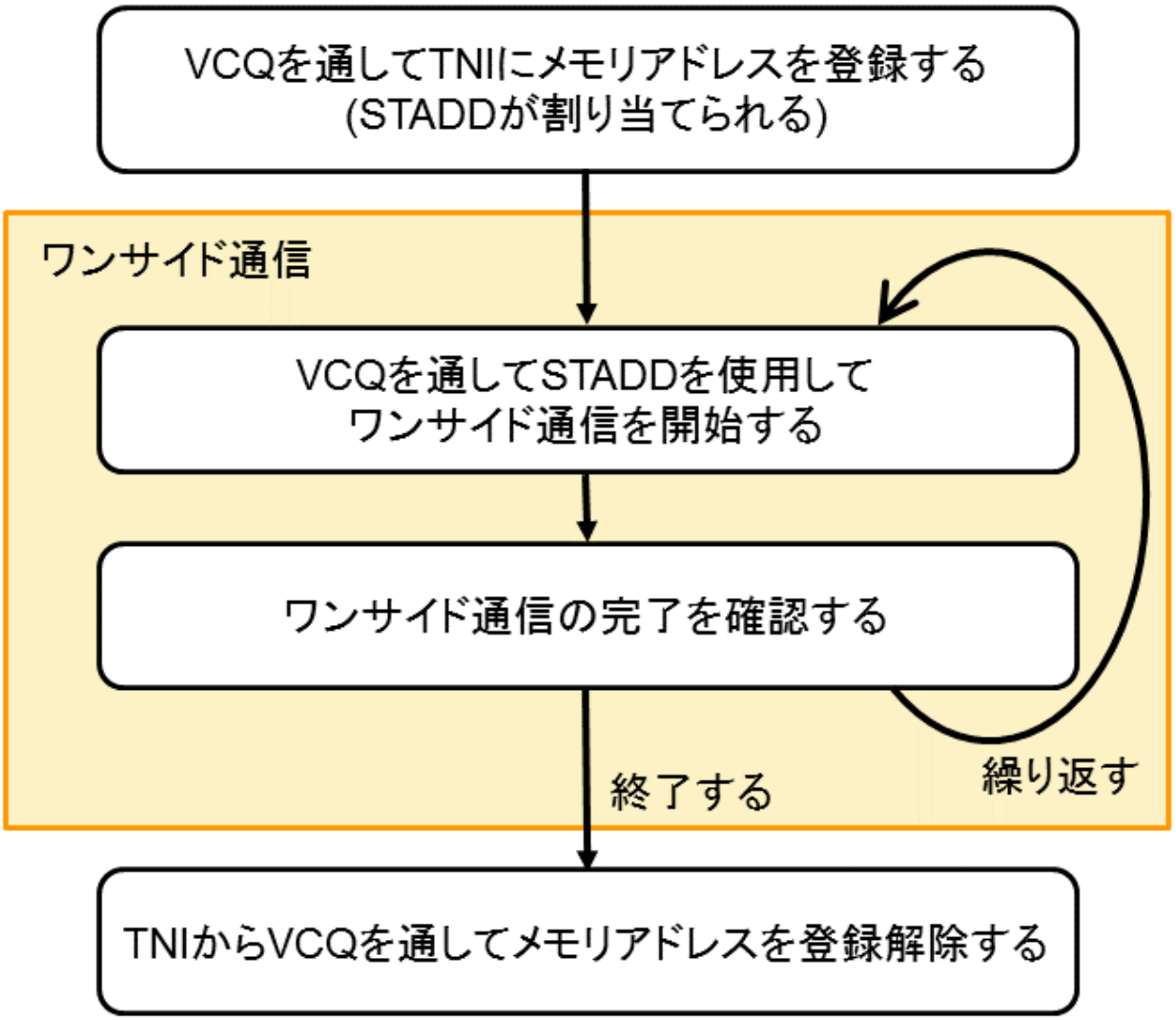
2.2.2 STADD(Steering Address)

通常、ユーザー空間のプロセスは仮想メモリアドレスでメモリ領域を識別しますが、Tofuインターコネクトは、OSを介在せずにメモリにアクセスするため、仮想メモリアドレスを認識しません。そのため、Tofuインターコネクトでは、ワンサイド通信による参照・更新の対象となるメモリ領域を、STag(steering tag)と呼ぶ値で識別します。

uTofuでは、STagを容易に扱うために、STADD(steering address)と呼ぶ値を導入しています。STADDは、64ビット符号なし整数であり、CQ内で一意の値を持ちます。同じメモリ領域でもCQが異なればSTADDの値も異なることがあります。

プロセスの使用できる仮想メモリに対してSTag(およびSTADD)は自動的には割り当たりません。そのため、ワンサイド通信を開始する前に、TNIに対してメモリアドレスを登録して対応するSTADDの値を得る必要があります。一度登録したメモリアドレスに対するSTADDは、複数回のワンサイド通信に使用できます。ワンサイド通信での使用が終了したら、TNIに対してその登録を解除する必要があります。登録の解除が行われずに多数のメモリ領域が登録されたままになっていると、STagの枯渇やuTofuのSTADD管理関数の速度低下を招くことがあります。

図2.2 STADDの登録から解除までの流れ



uTofuでワンサイド通信を行う場合は、uTofuの関数の引数に、ローカルとリモートのSTADDを与えます。そのため、通信を開始する前に、リモートのSTADDを知っておく必要があります。

STADDはVCQごとに管理されます。

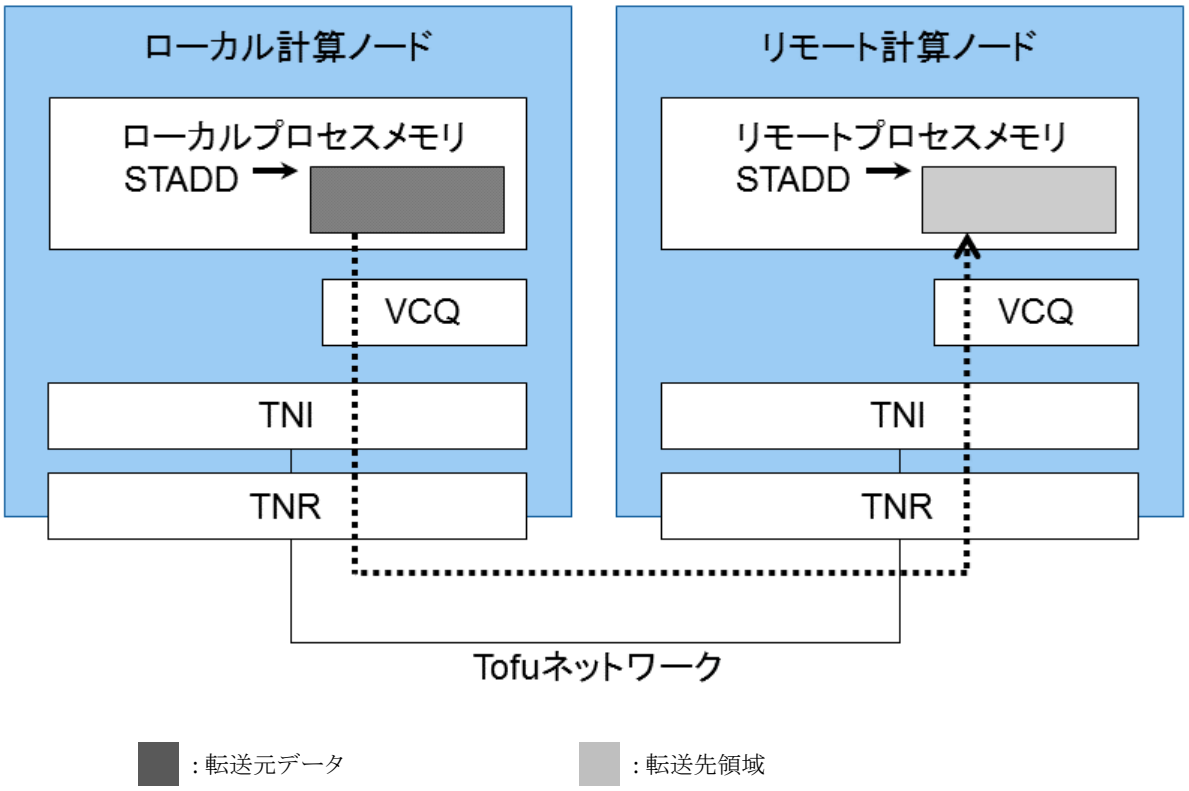
2.2.3 Put通信

Putは、ローカルプロセスのメモリ(転送元)にある連続データをリモートプロセスのメモリ(転送先)に書き込む通信機能です。

ローカル計算ノードで使用するTNIはVCQハンドルで指定します。リモート計算ノードで使用するTNIはリモートVCQ IDで指定します。転送元のメモリの開始アドレスはローカルSTADDで指定します。転送先のメモリの開始アドレスはリモートSTADDで指定します。

Tofuインターコネクトが行うPutの通信モデルを下図に示します。この図では、点線の矢印がデータの流れを表しています。

図2.3 Putの通信モデル



Putは以下の手順で行われます。

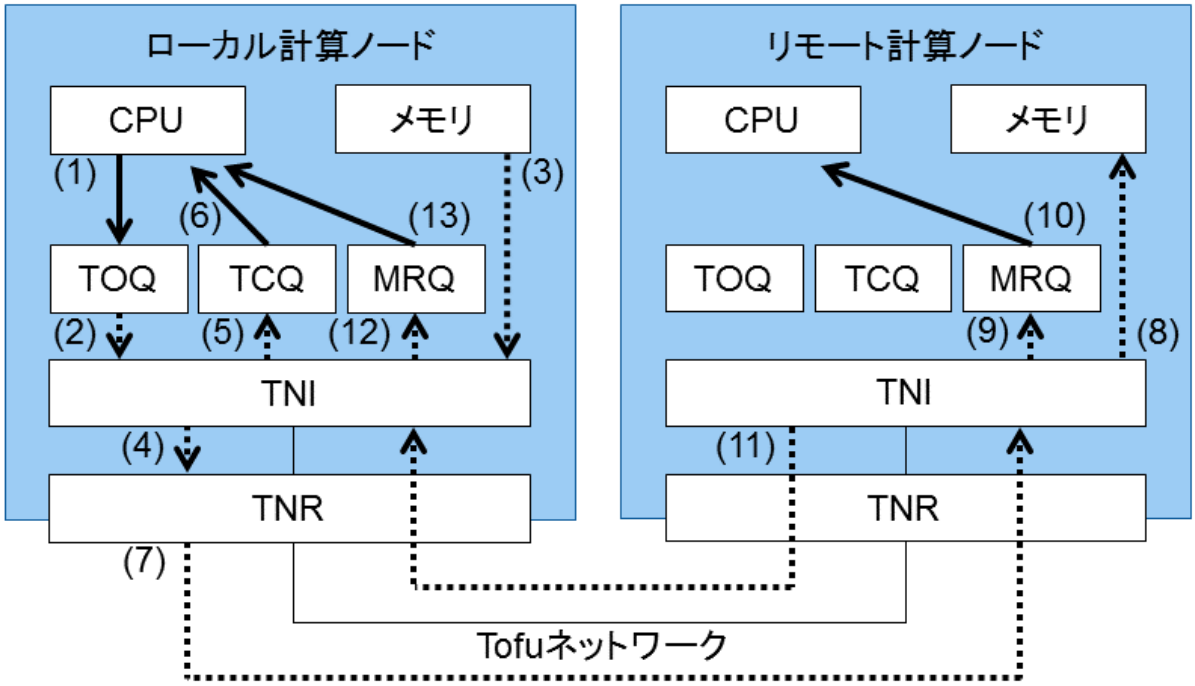
このうち1、6、10、および13は、uTofu関数を呼び出して行う必要があります。

1. ローカルプロセスがTOQにディスクリプタを書き込み、TNIに対して通信開始の指示を行う。
2. ローカルTNIがTOQからディスクリプタを読み込む。
3. ローカルTNIがメモリからデータを読み込む。
4. ローカルTNIがTNRにデータを送出する。
5. ローカルTNIがTCQにディスクリプタを書き込む。
6. ローカルプロセスがTCQからディスクリプタを読み込む。ローカルプロセスは、このTCQディスクリプタを確認することで、これ以降に転送元のメモリ領域を書き換えても転送先に書き込まれるデータには影響しないと判断できる。
7. TNRとTofuネットワークがデータをリモートTNIに転送する。
8. リモートTNIがデータをメモリに書き込む。
9. リモートTNIがMRQにリモート通知ディスクリプタを書き込む。
10. リモートプロセスがMRQからディスクリプタを読み込む。リモートプロセスは、このリモート通知MRQディスクリプタを確認することで、転送先のメモリ領域が書き換わったことを知ることができる。
11. リモートTNIがローカルTNIにTNRとTofuネットワークを経由してメモリ書込みの完了を通知するパケットを転送する。
12. ローカルTNIがMRQにローカル通知ディスクリプタを書き込む。
13. ローカルプロセスがMRQからディスクリプタを読み込む。ローカルプロセスは、このローカル通知MRQディスクリプタを確認することで、転送先のメモリ領域が書き換わったことを知ることができる。

これらのTCQディスクリプタ、リモート通知MRQディスクリプタ、ローカル通知MRQディスクリプタは、ローカルプロセスがTOQにディスクリプタを書き込むときに、通知をする・しないを選択できます。

これらの手順を下図に示します。この図の括弧内の数字が上記の手順に該当します。実線の矢印は、uTofuの関数を呼び出して行う処理を表しています。

図2.4 Putの手順



Putでは、ローカルTNIは、TOQディスクリプタを読み込んだ後、メモリからデータを読み込みます。Tofuインターコネクトでは、データサイズが小さい場合に、ローカルプロセスがデータをTOQディスクリプタに埋め込むこともできます。これをピギーバックと呼びます。ピギーバックを使うと、ローカルTNIがメモリからデータを読み込む処理がなくなり、低遅延な通信を行うことができます。また、ローカルプロセスでメモリ領域をTNIに登録しておく必要もありません。ピギーバックを使用可能なデータサイズはuTofuで問い合わせることができます。

Put通信のプログラム例については、“5.1.1 Putによるping-pong通信の例”を参照してください。

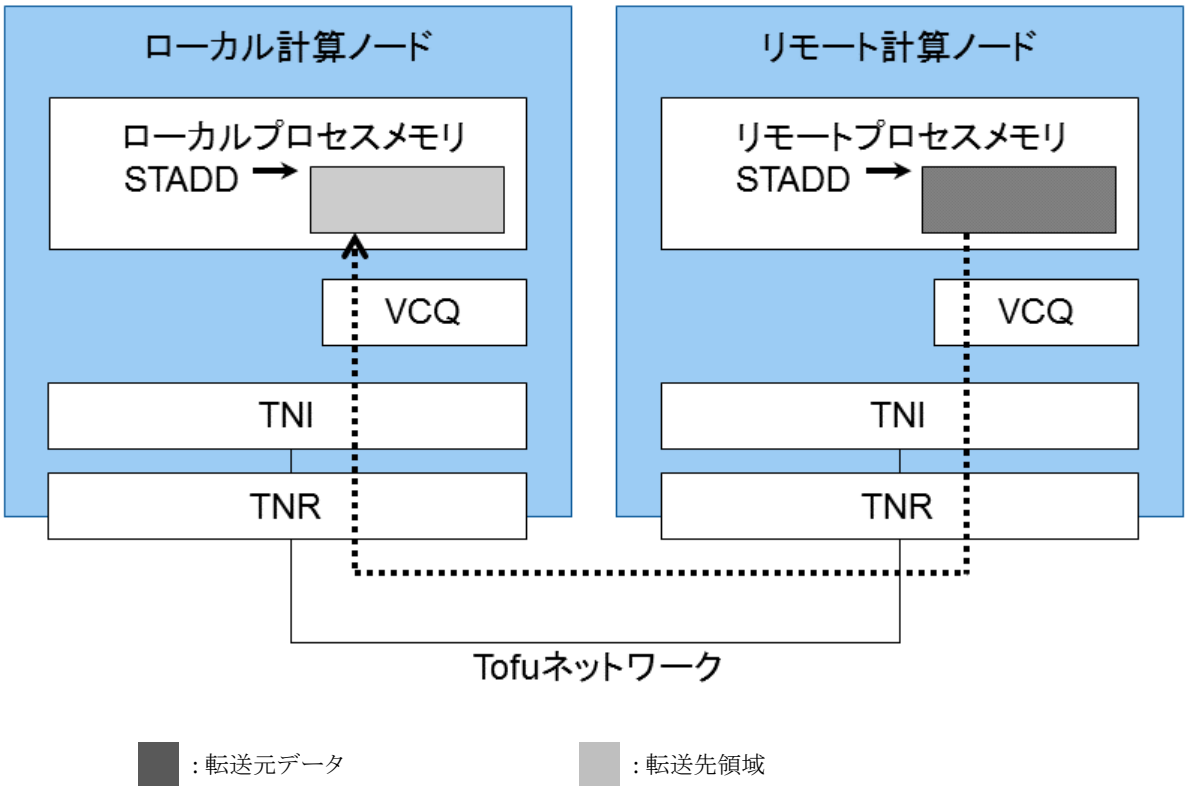
2.2.4 Get通信

Getは、リモートプロセスのメモリ(転送元)にある連続データをローカルプロセスのメモリ(転送先)に書き込む通信機能です。

ローカル計算ノードで使用するTNIはVCQハンドルで指定します。リモート計算ノードで使用するTNIはリモートVCQ IDで指定します。転送元のメモリの開始アドレスはリモートSTADDで指定します。転送先のメモリの開始アドレスはローカルSTADDで指定します。

Tofuインターコネクトが行うGetの通信モデルを下図に示します。この図では、点線の矢印がデータの流れを表しています。

図2.5 Getの通信モデル



Getは以下の手順で行われます。

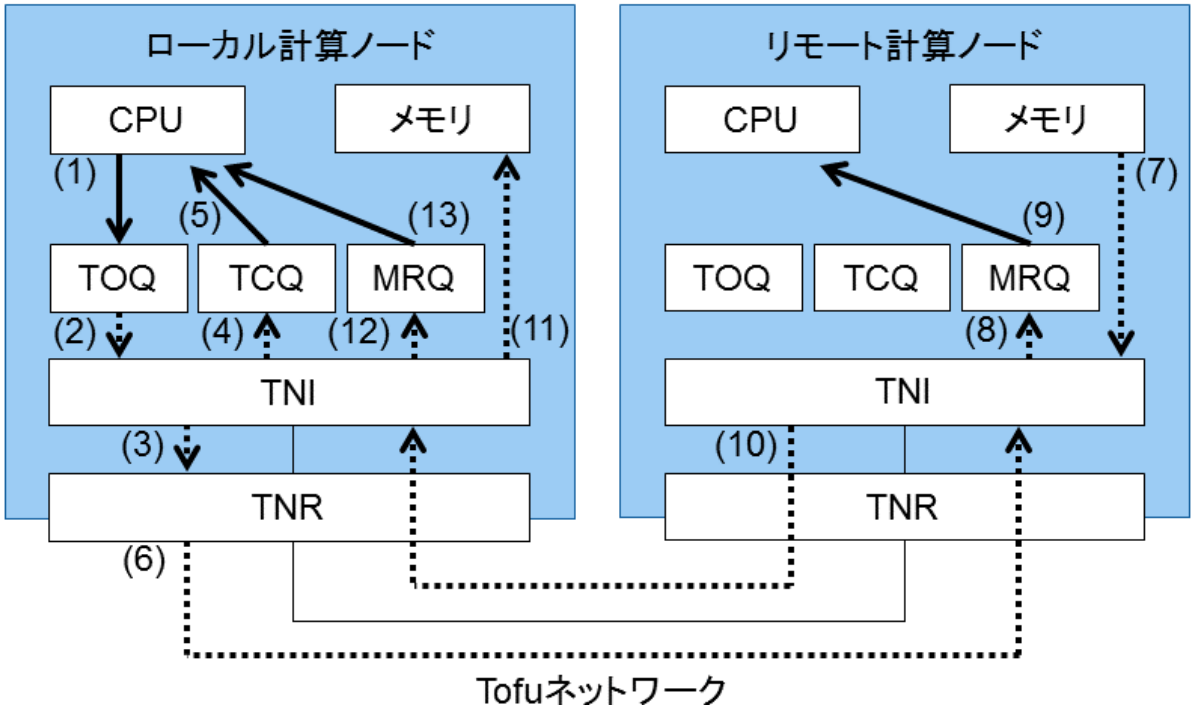
このうち1、5、9、および13は、uTofu関数を呼び出して行う必要があります。

1. ローカルプロセスがTOQにディスクリプタを書き込み、TNIに対して通信開始の指示を行う。
2. ローカルTNIがTOQからディスクリプタを読み込む。
3. ローカルTNIがTNRにデータ転送を指示するパケットを送出する。
4. ローカルTNIがTCQにディスクリプタを書き込む。
5. ローカルプロセスがTCQからディスクリプタを読み込む。ローカルプロセスは、このTCQディスクリプタを確認することで、通信が開始されたことを知ることができる。
6. TNRとTofuネットワークが指示パケットをリモートTNIに転送する。
7. リモートTNIがメモリからデータを読み込む。
8. リモートTNIがMRQにリモート通知ディスクリプタを書き込む。
9. リモートプロセスがMRQからディスクリプタを読み込む。リモートプロセスは、このリモート通知MRQディスクリプタを確認することで、これ以降に転送元のメモリ領域を書き換えても転送先に書き込まれるデータには影響しないと判断できる。
10. リモートTNIがローカルTNIにTNRとTofuネットワークを経由してデータを転送する。
11. ローカルTNIがデータをメモリに書き込む。
12. ローカルTNIがMRQにローカル通知ディスクリプタを書き込む。
13. ローカルプロセスがMRQからディスクリプタを読み込む。ローカルプロセスは、このローカル通知MRQディスクリプタを確認することで、転送先のメモリ領域が書き換わったことを知ることができる。

これらのTCQディスクリプタ、リモート通知MRQディスクリプタ、ローカル通知MRQディスクリプタは、ローカルプロセスがTOQにディスクリプタを書き込むときに、通知をする・しないを選択できます。

これらの手順を下図に示します。この図の括弧内の数字が上記の手順に該当します。実線の矢印は、uTofuの関数を呼び出して行う処理を表しています。

図2.6 Getの手順



Get通信のプログラム例については、“5.1.2 Getによる状況確認の例”を参照してください。

2.2.5 ARMW(Atomic Read Modify Write)通信

ARMWは、リモート計算ノードの主記憶上の4バイトまたは8バイトのデータに対して読込み、演算、書込みを不可分に行う通信機能です。

uTofuで使用可能な演算種別を下表に示します。

表2.2 ARMWの演算種別

名称	演算
CSWAP	コンペア・アンド・スワップ
SWAP	スワップ
ADD	符号なし整数として加算
XOR	ビットごとの排他的論理和
AND	ビットごとの論理積
OR	ビットごとの論理和

CSWAPでは、以下のように処理が行われます。

1. ローカルプロセスは、書込みオペランドと比較オペランドの2つをTOQディスクリプタに指定します。
2. リモート計算ノードのTNIは、対象メモリにおいて、演算実行前の値が比較オペランドと同じ場合だけ書込みオペランドの値を書き込みます。

SWAPでは、以下のように処理が行われます。

1. ローカルプロセスは、1つの書込みオペランドをTOQディスクリプタに指定します。
2. リモート計算ノードのTNIは、対象メモリにおいて、演算実行前の値にかかわらず書込みオペランドの値を書き込みます。

それ以外の演算種別では、以下のように処理が行われます。

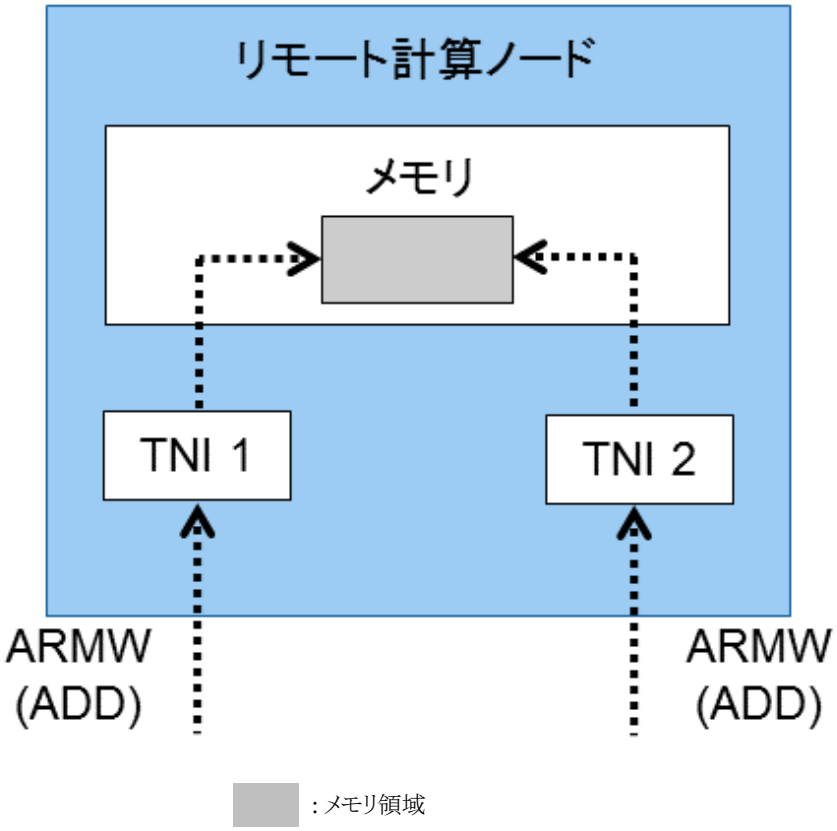
1. ローカルプロセスは、1つのオペランドをTOQディスクリプタに指定します。
2. リモート計算ノードのTNIは、対象メモリにおいて、演算実行前の値とオペランドとの演算を行って、その結果を書き込みます。

どの演算の場合も、演算実行前の値がローカル通知MRQディスクリプタにより通知されます。例えば、ADDでは、ローカルプロセスがオペランドとして値11を指定し、リモート計算ノードの対象メモリの値が22であった場合、その対象メモリには値33が書き込まれ、ローカルプロセスにはローカル通知MRQディスクリプタにより値22が通知されます。

ARMWでは、リモート計算ノードでのメモリ書換え操作が不可分に行われます。不可分とは、2つ以上のメモリ書換え操作がほぼ同時に行われるときに、1つのメモリ書換え操作による読込み、演算、および書込みの一連の処理の実行中に、ほかのメモリ書換え操作による読込み、演算、または書込みのいずれの処理も実行されることがないことを意味します。この不可分性は、同一のリモートVCQによるARMWの実行だけでなく、ほかのリモートVCQやほかのリモートTNIによるARMWの実行、および、リモートCPUによる不可分なメモリ書換え命令の実行に対しても、保証されます。

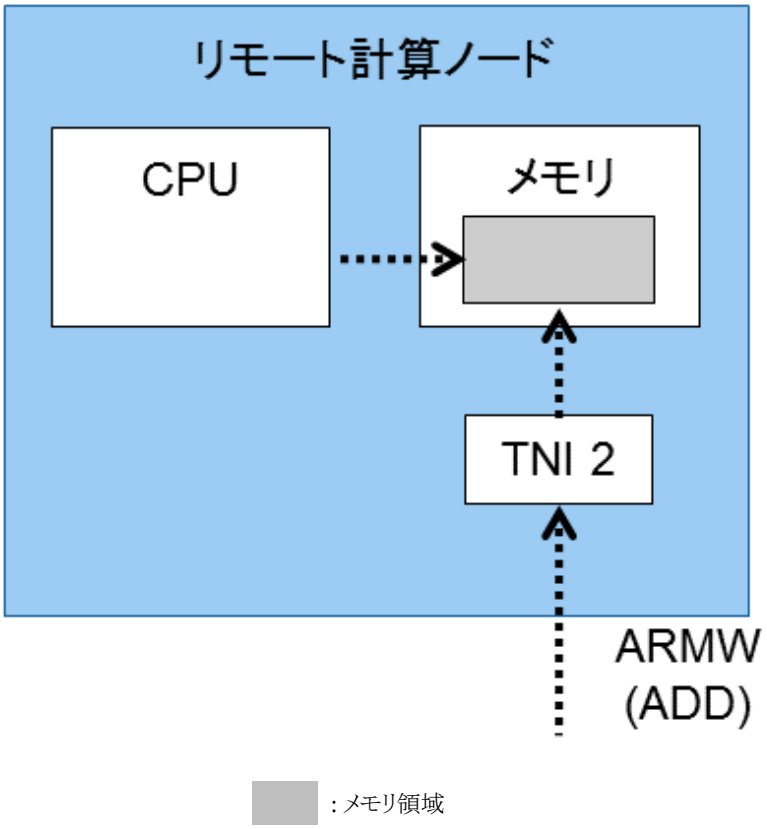
例えば、2つの計算ノードから別の計算ノードの同じメモリ領域に、同時にADD演算のARMW通信を実行した場合を考えます。使用したVCQの組合せに関わらず、まずは片方の読込み、演算、および書込みが行われます。次に他方の読込み、演算、および書込みが行われます。その結果、そのメモリ領域の最終的な値は、2つのオペランドと元の値の3つの和になります。

図2.7 同じメモリ領域に、同時にADD演算のARMW通信を実行した場合



同様に、ある計算ノードから別の計算ノードにADD演算のARMW通信を実行し、それと同時にそのリモート計算ノードのCPUで同じメモリ領域に対して不可分な加算の命令を実行した場合も、2組の処理が順に行われ、そのメモリ領域の最終的な値は3つの値の和になります。

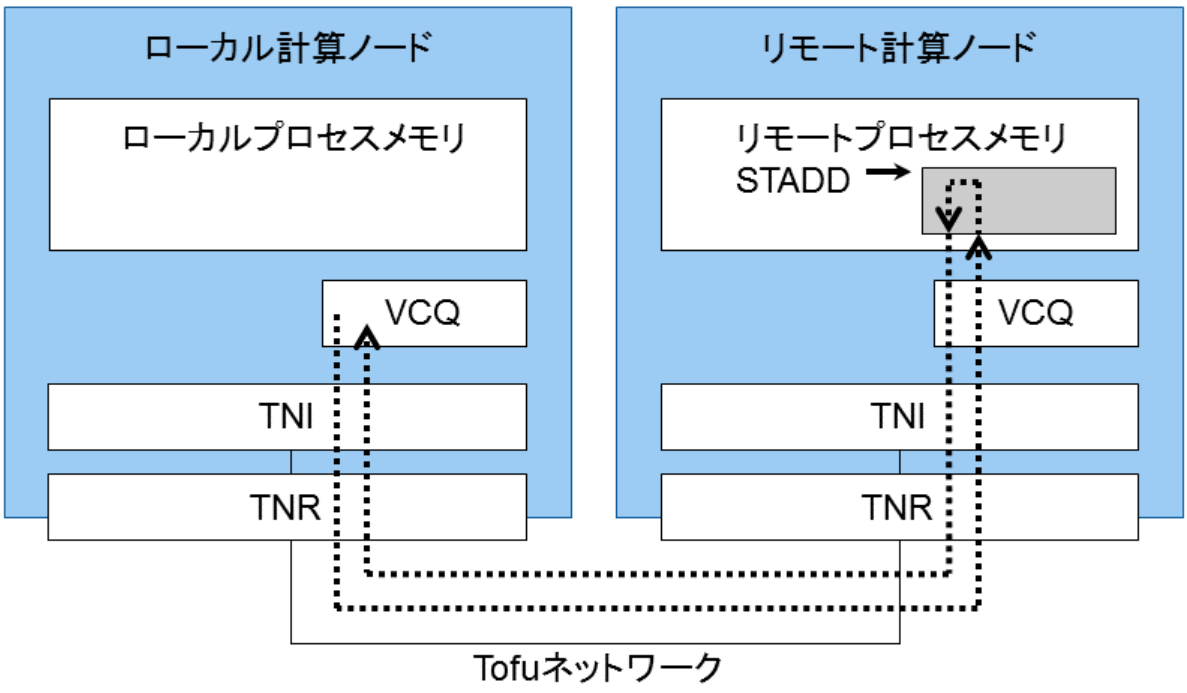
図2.8 ADD演算のARMW通信とCPUが同じメモリ領域に対して不可分な加算の命令を実行した場合



ローカル計算ノードで使用するTNIはVCQハンドルで指定します。リモート計算ノードで使用するTNIはリモートVCQ IDで指定します。対象メモリの開始アドレスはリモートSTADDで指定します。

Tofuインターコネクトが行うARMWの通信モデルを下図に示します。この図では、点線の矢印がデータの流れを表しています。

図2.9 ARMWの通信モデル



: 演算対象領域

ARMWは以下の手順で行われます。

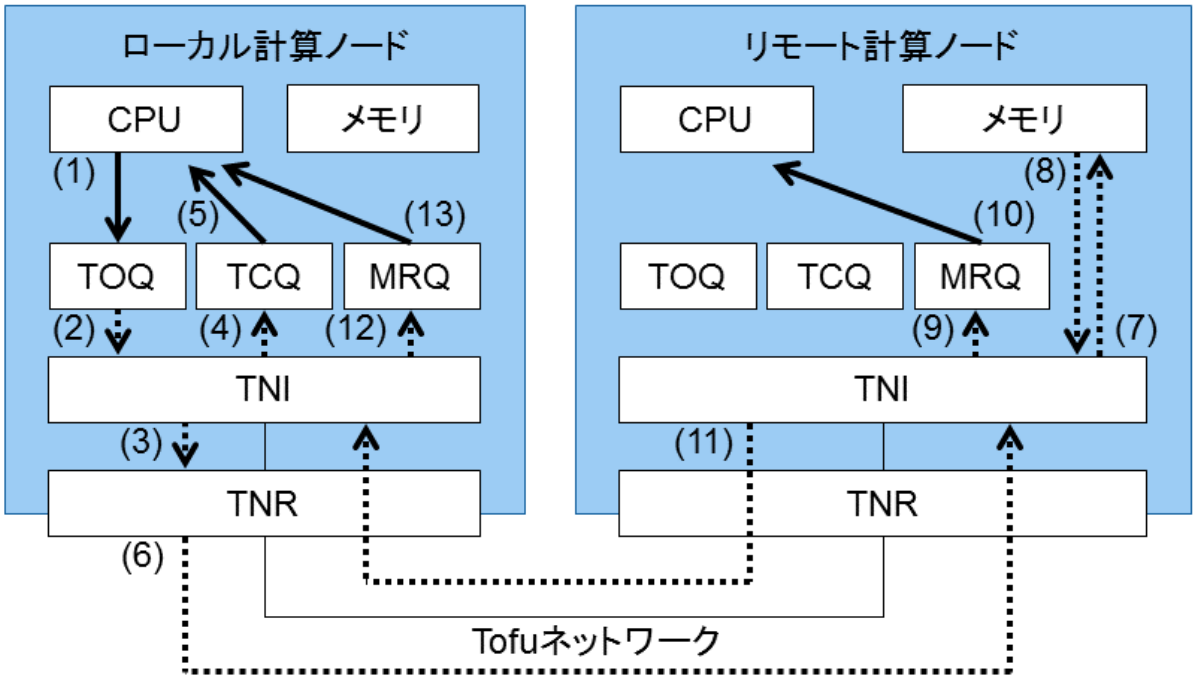
このうち1、5、10、および13は、uTofu関数を呼び出して行う必要があります

1. ローカルプロセスがTOQにディスクリプタを書き込み、TNIに対して通信開始の指示を行う。
2. ローカルTNIがTOQからディスクリプタを読み込む。
3. ローカルTNIがTNRに演算種別とオペランドを送出する。
4. ローカルTNIがTCQにディスクリプタを書き込む。
5. ローカルプロセスがTCQからディスクリプタを読み込む。ローカルプロセスは、このTCQディスクリプタを確認することで、通信が開始されたことを知ることができる。
6. TNRとTofuネットワークが演算種別とオペランドをリモートTNIに転送する。
7. リモートTNIとメモリコントローラが、メモリからデータを読み込み、演算を行い、メモリに演算後のデータを書き込む。
8. リモートTNIが演算前のデータを取得する。
9. リモートTNIがMRQにリモート通知ディスクリプタを書き込む。
10. リモートプロセスがMRQからディスクリプタを読み込む。リモートプロセスは、このリモート通知MRQディスクリプタを確認することで、演算対象のメモリ領域が書き換わったことを知ることができる。
11. リモートTNIがローカルTNIにTNRとTofuネットワークを経由してデータを転送する。
12. ローカルTNIがMRQにローカル通知ディスクリプタを書き込む。
13. ローカルプロセスがMRQからディスクリプタを読み込む。ローカルプロセスは、このローカル通知MRQディスクリプタを確認することで、演算対象のメモリ領域が書き換わったことを知ることができる。

これらのTCQディスクリプタ、リモート通知MRQディスクリプタ、ローカル通知MRQディスクリプタは、ローカルプロセスがTOQにディスクリプタを書き込むときに、通知をする・しないを選択できます。

これらの手順を下図に示します。この図の括弧内の数字が上記の手順に該当します。実線の矢印は、uTofuの関数を呼び出して行う処理を表しています。

図2.10 ARMWの手順



ARMW(Atomic Read Modify Write)通信のプログラム例については、“5.1.3 ARMWによるゲームの例”を参照してください。

2.2.6 NOP

NOPは、何の通信もしません。通常は“2.2.8 セッションモード”に示すセッションモードでTOQディスクリプタの数合わせをするために使用します。“2.2.7 フリーモード”に示すフリーモードで使用することもできますが、TOQの枠を無駄に消費する以外の効果はありません。

NOPは以下の手順で行われます。MRQにディスクリプタが書き込まれることはありません。

1. ローカルプロセスがTOQにディスクリプタを書き込み、TNIに対して処理開始の指示を行う。
2. ローカルTNIがTOQからディスクリプタを読み込む。
3. ローカルTNIがTCQにディスクリプタを書き込む。
4. ローカルプロセスがTCQからディスクリプタを読み込む。

このTCQディスクリプタは、ローカルプロセスがTOQにディスクリプタを書き込むときに、通知をする・しないを選択できます。

2.2.7 フリーモード

CQには以下の2つの動作モードがあり、各CQにはどちらか一方のモードがあらかじめ割り当てられています。VCQの作成時に、どちらのモードのCQにVCQを作成するかを指定することができます。1つのVCQで両方のモードを同時に使用することはできません。

- フリーモード
- セッションモード

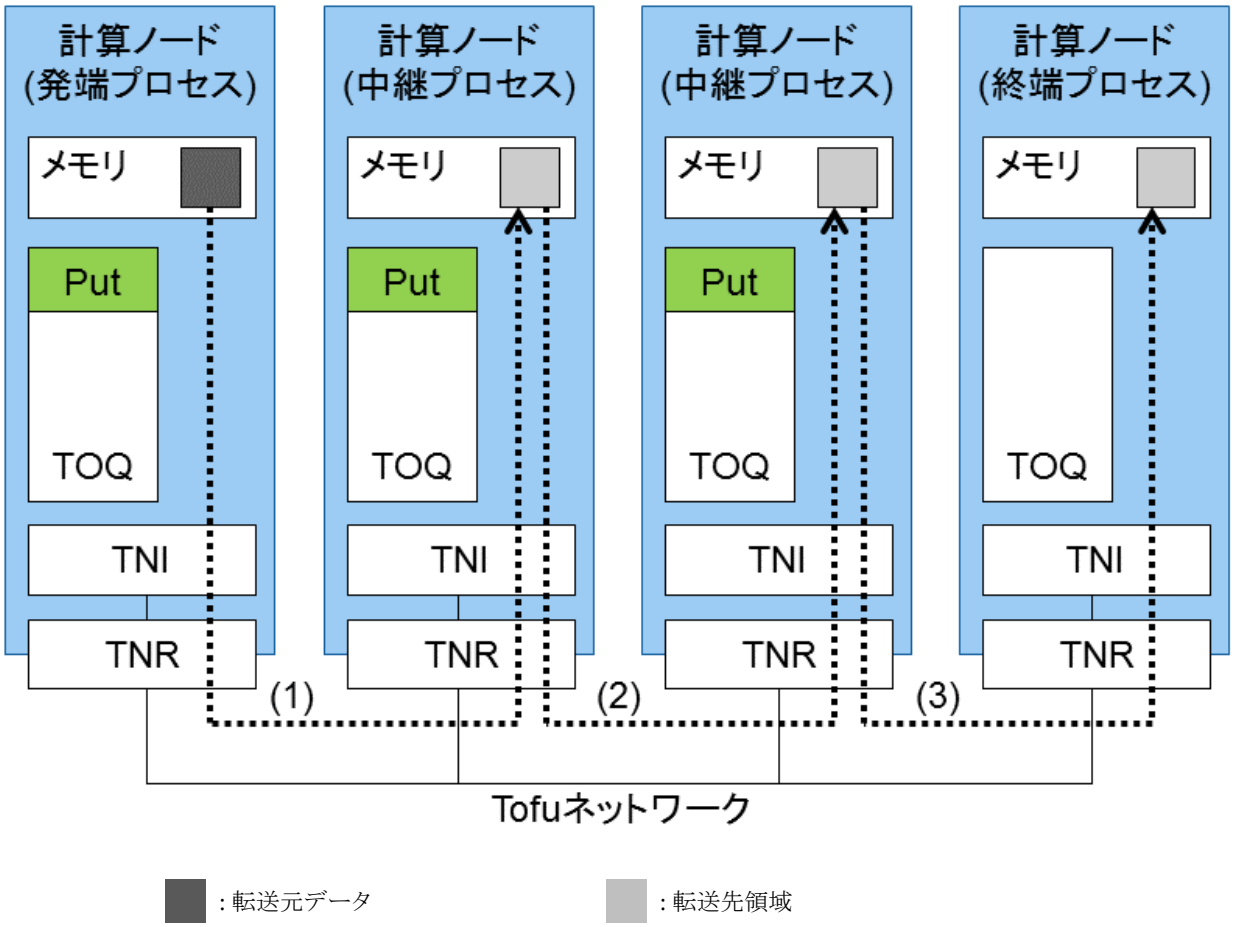
フリーモードは通常のモードであり、“2.2.3 Put通信”から“2.2.6 NOP”に示した動作はフリーモードを想定した動作です。フリーモードでは、uTofuのワンサイド通信開始関数またはワンサイド通信一括開始関数を呼び出すことで、通信指示のディスクリプタがTOQに書き込まれ、TNIが通信を開始します。したがって、2つ以上のプロセスが連携して一連の通信を行うときに、適切なタイミングでプロセスが関数を呼び出す必要があります。

フリーモードのCQに作成されたVCQをフリーモードVCQと呼びます。VCQの作成時にモードを指定しなければ、フリーモードVCQが作成されます。

2.2.8 セッションモード

セッションモードは、ほかの通信と連動して、あらかじめ指示した通信を自動的に開始するモードです。セッションモードでは、uTofuのワンサイド通信開始関数またはワンサイド通信一括開始関数を呼び出すことで、通信指示のディスクリプタがTOQに書き込まれます。しかしそれだけではTNIは通信を開始しません。指示したVCQにほかのPut通信が届くことで、TNIは指示されていた通信を自動的に開始します。したがって、2つ以上のプロセスが連携して一連の通信を行うときに、プロセスの関与なしに2つ目以降の通信を開始することができます。例えば、あるプロセスが1つのデータを2つ以上のプロセスに送信するときに、下図のようにデータの中継を自動的に行うことができます。

図2.11 4プロセスでの直列Put中継



セッションモードのCQに作成されたVCQをセッションモードVCQと呼びます。VCQの作成時にセッションモードを指定することにより、セッションモードVCQが作成されます。

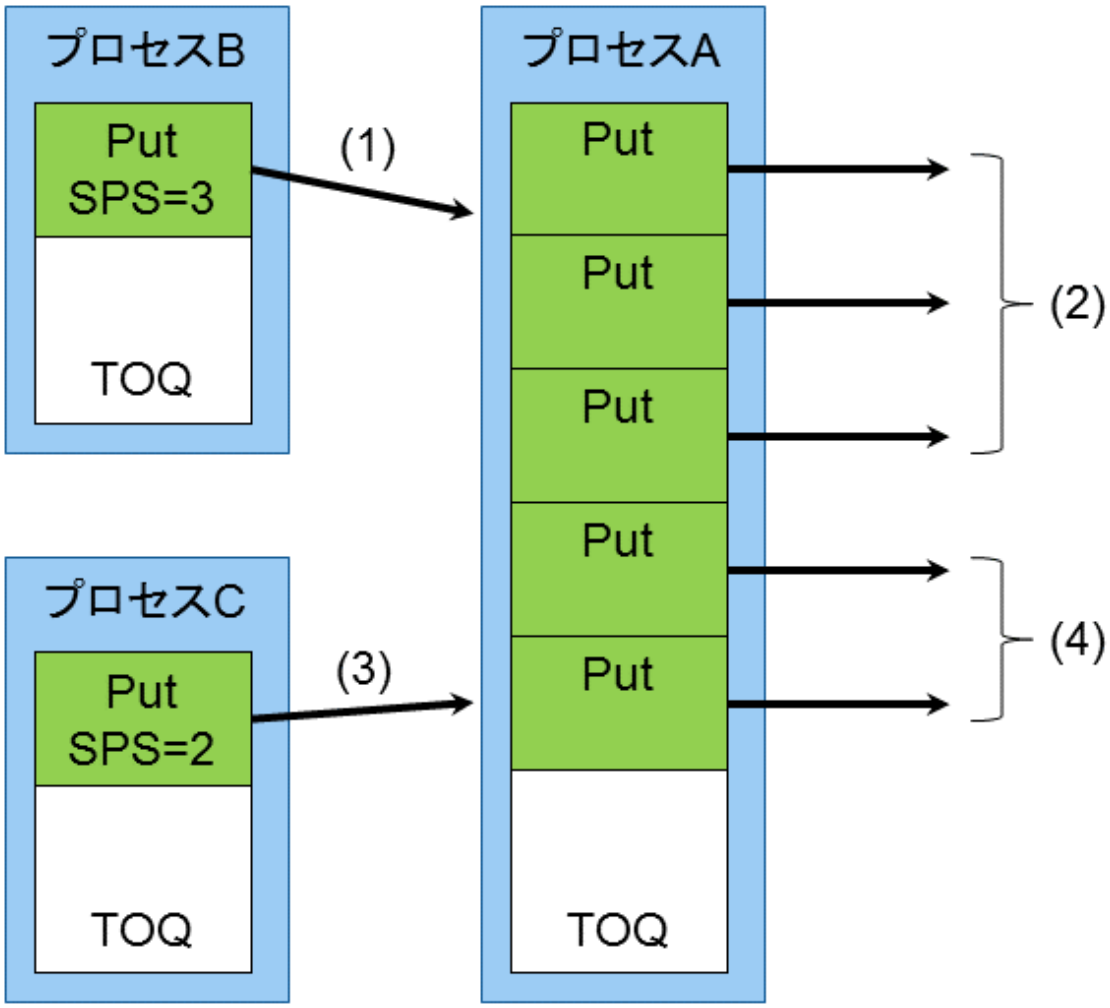
この項では、説明の便宜上、下表に示す用語を用います。

用語	説明
発端プロセス	一連の通信の中で最初にリモートVCQとしてセッションモードVCQを指定して送信するプロセス
中継プロセス	セッションモードVCQを使用して自動的に受信・送信するプロセス
終端プロセス	中継プロセスのセッションモードVCQからの通信を受信するが送信しないプロセス

2.2.8.1 セッションモードの動作の詳細

セッションモードでは、Put通信のリモートプロセス(通信の標的側)のTNIで自動的に開始する通信の数を、ローカルプロセス(通信の指示側)で指定できます。この数をセッションオフセット前進量(SPS: session progress step)と呼びます。例えば、プロセスAであらかじめセッションモードVCQのTOQに5つのディスクリプタを書いておき、SPSに3を指定したプロセスA宛てのPut通信をプロセスBが指示すると、そのPut通信がプロセスAに届いたときに、最初の3つのTOQディスクリプタによる通信が自動的に開始されます。続けてSPSに2を指定したプロセスA宛てのPut通信をプロセスCが指示すると、そのPut通信がプロセスAに届いたときに、次の2つのTOQディスクリプタによる通信が自動的に開始されます。この様子を下図に示します。

図2.12 SPSの指定



ローカルプロセスでSPSを指定できるディスクリプタはPutだけです。Put Piggyback、Get、ARMW、またはNOPディスクリプタにSPSを指定した場合の動作は保証されません。リモートプロセスで自動的に開始できるディスクリプタはPutとNOPだけです。セッションモードVCQにPut Piggyback、Get、またはARMWのディスクリプタを指定した場合の動作は保証されません。

SPSを指定したディスクリプタは、フリーモードVCQとセッションモードVCQのどちらにでも使用できます。発端プロセスでは、通常、フリーモードVCQを使用します。中継プロセスではセッションモードVCQを使用します。終端プロセスではフリーモードVCQまたはセッションモードVCQを使用します。終端プロセスでセッションモードVCQを使用する場合は、そのVCQに送信するPut通信ではSPSを0にする必要があります。

中継プロセスでは、1つ前の発端プロセスまたは中継プロセスから受信したPut通信のデータがメモリに書き込まれた後に次の中継プロセスまたは終端プロセスへのPut通信を開始することが保証されます。したがって、受信したPut通信のリモートSTADDに指定されていた値と送信するPut通信のローカルSTADDに指定した値が同じであった場合に、受信したデータを送信することが保証されます。ただし、それとほぼ同時にほかの通信またはCPUによって該当のメモリ領域が書き換えられた場合は、その限りではありません。

リモートSTADDの値が異常などの原因により、届いたPut通信がエラーとなった場合は、そのPut通信ではセッションモードVCQに指示した通信が開始されません。

セッションモードVCQへのTOQディスクリプタの書き込みを、Put通信が届いた後に行うこともできます。セッションモードVCQにPut通信が届いたときに、まだ開始していないTOQディスクリプタがあり、かつ、その数がPut通信のSPSの値より少ない場合は、まず今ある分のTOQディスクリプタによる通信が開始されます。その後、そのセッションモードVCQを指定したワンサイド通信開始関数またはワンサイド通信一括開始関数が呼ばれたときに、不足していた分のTOQディスクリプタによる通信が順次開始されます。2つ以上のPut通信にまたがってTOQディスクリプタが不足している場合も同様です。しかし、TOQディスクリプタの不足数は2000までが許容されます。2000を超えると、ワンサイド通信開始関数またはワンサイド通信一括開始関数を呼んでもTOQディスクリプタによる通信が開始されないなど、通信が正常に行われない場合があります。

SPSには0から15までの値を指定できます。16以上の値を指定した場合の動作は保証されません。

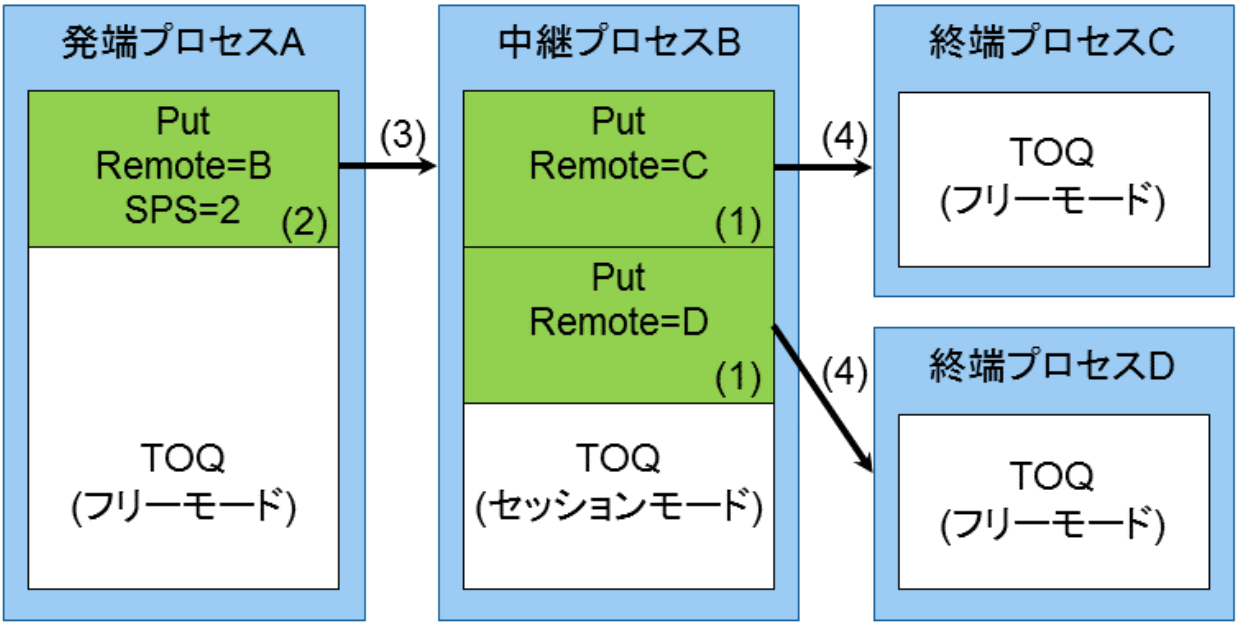
TCQやMRQへのディスクリプタの書込みなど、ほかの動作はフリーモードと同じです。

2.2.8.2 セッションモードによる通信の分岐の例

中継プロセスでは、Put通信を1つ1つ中継するだけではなく、分岐しながら中継することができます。

中継プロセスでPut通信を複数のプロセスに分岐する場合の例を下図に示します。

図2.13 中継プロセスでの分岐



この例では以下の流れでPut通信の分岐が行われます。図の括弧内の数字が以下の手順に該当します。

1. 中継プロセスBで、終端プロセスC宛てと終端プロセスD宛ての2つのPutディスクリプタを書いておく。
2. 発端プロセスAで、SPSを2とした中継プロセスB宛てのPut通信を指示する。
3. 2.により、発端プロセスAで2.のPut通信が開始される。
4. 3.のPut通信が中継プロセスBに届くと、1.の2つのPut通信が自動的に開始される。

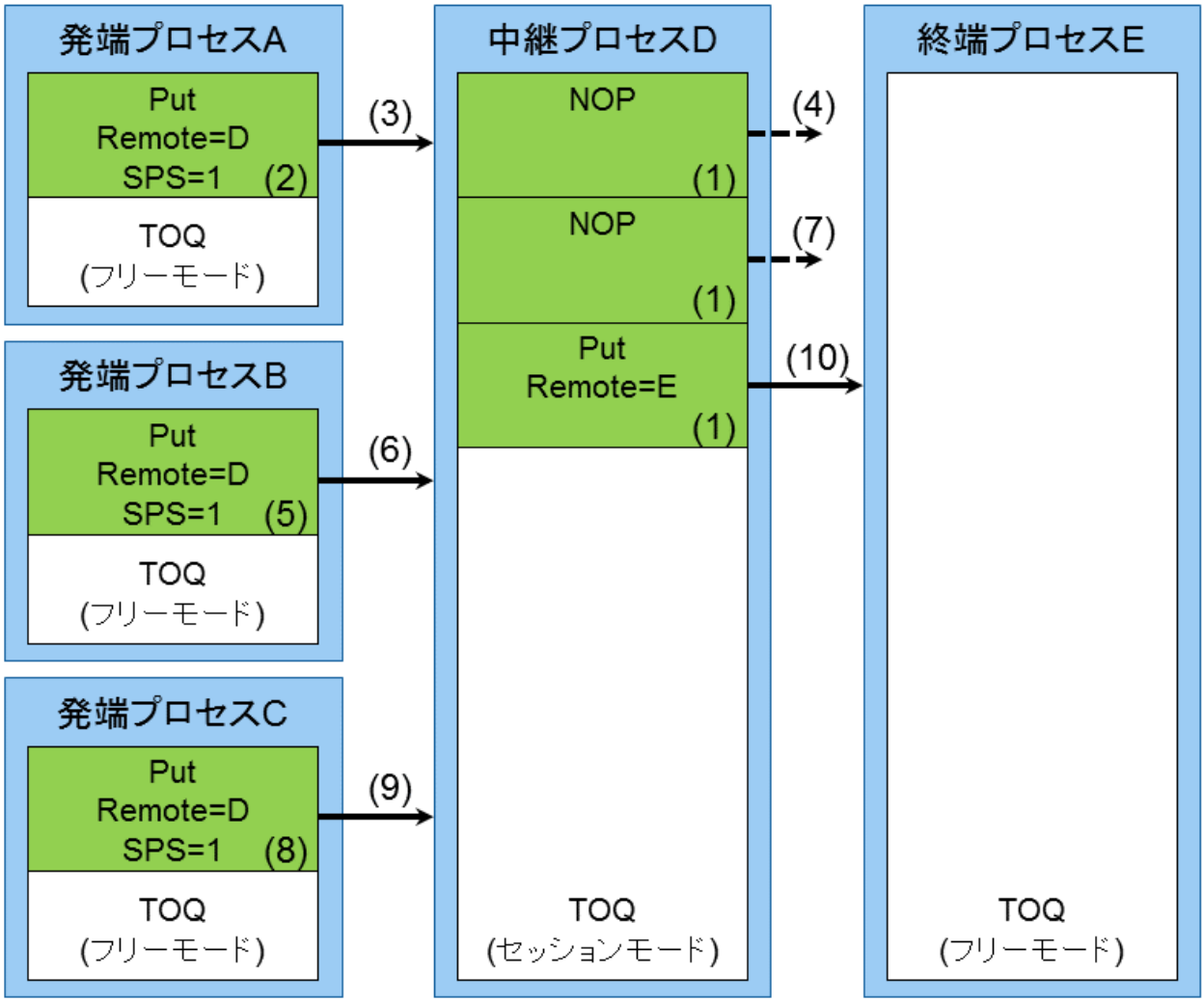
この例では中継プロセスはBだけですが、プロセスCとプロセスDも中継プロセスとすることで、複数の段階で分岐することができます。

2.2.8.3 セッションモードによる通信の待ち合わせの例

さらに、中継プロセスでは、待ち合わせながら中継することができます。

中継プロセスで3つのプロセスからのPut通信を待ち合わせる場合の例を下図に示します。

図2.14 中継プロセスでの待ち合わせ



この例では以下の流れでPut通信の待ち合わせが行われます。図の括弧内の数字が以下の手順に該当します。

1. 中継プロセスDで、2つのNOPディスクリプタと終端プロセスE宛ての1つのPutディスクリプタを書いておく。
2. 発端プロセスAで、SPSを1とした中継プロセスD宛てのPut通信を指示する。
3. 2.により、発端プロセスAで2.のPut通信が開始される。
4. 3.のPut通信が中継プロセスDに届くと、1.の1つ目のNOPが自動的に開始されるが、通信は発生しない。
5. 発端プロセスBで、SPSを1とした中継プロセスD宛てのPut通信を指示する。
6. 5.により、発端プロセスBで5.のPut通信が開始される。
7. 6.のPut通信が中継プロセスDに届くと、1.の2つ目のNOPが自動的に開始されるが、通信は発生しない。
8. 発端プロセスCで、SPSを1とした中継プロセスD宛てのPut通信を指示する。
9. 8.により、発端プロセスCで8.のPut通信が開始される。
10. 9.のPut通信が中継プロセスDに届くと、1.のPut通信が自動的に開始される。

この例では、発端プロセスA、B、CからのPut通信が、その順に指示されてその順に中継プロセスDに届くことになっています。実際には、順序は関係なく、3つのPut通信がすべて中継プロセスDに届いたタイミングで、次のPut通信が開始されます。

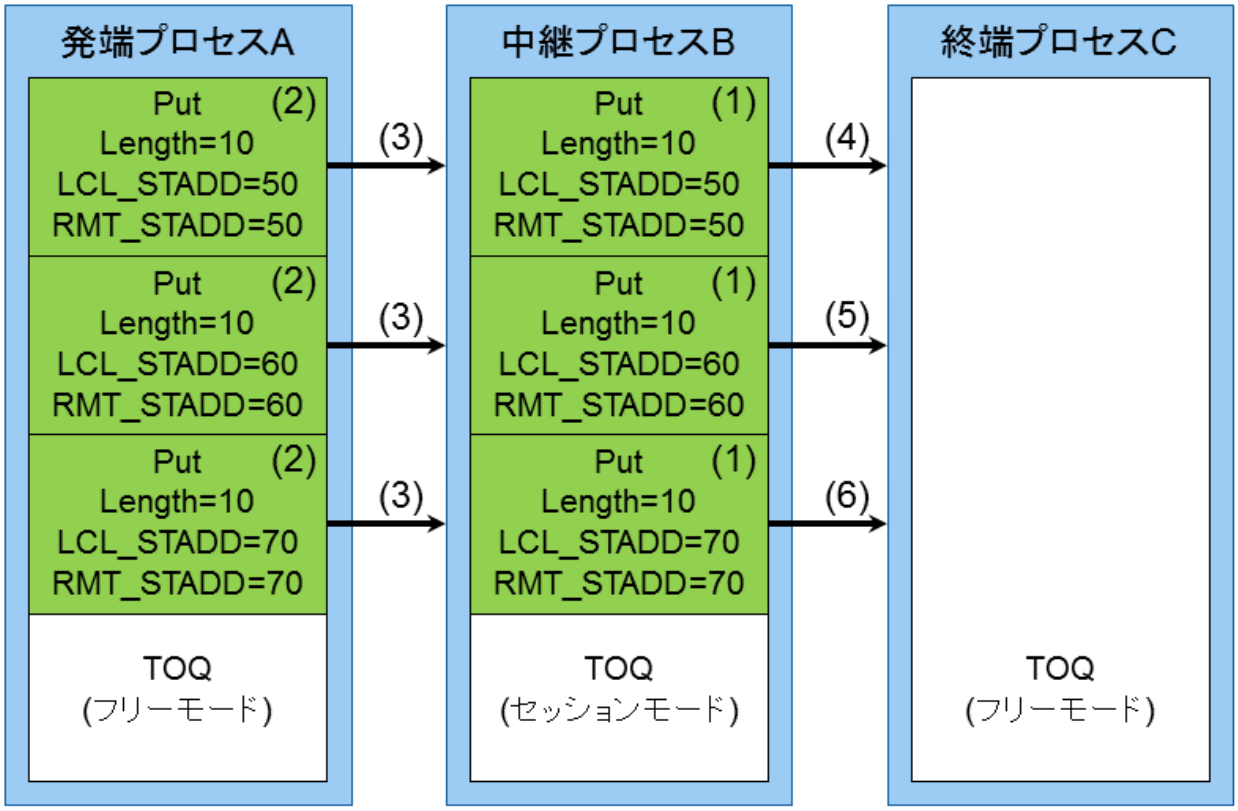
また、この例では中継プロセスはDだけですが、プロセスA、B、Cも中継プロセスとすることで、複数の段階で待ち合わせることができます。

2.2.8.4 セッションモードによる通信のパイプラインの例

データを1つのPut通信で中継すると、1つ前の発端プロセスまたは中継プロセスからデータがすべて届いてから、次の中継プロセスまたは終端プロセスへのPut通信を開始することになります。データのサイズが大きい場合は、データを分割した複数のPut通信によって、届いたデータから順次中継した方が、終端プロセスにすべてのデータが届くまでの時間を短くできる場合があります。このような中継方式をパイプラインと呼びます。

30バイトのデータを3つに分割してパイプライン化したPut通信で中継する場合の例を下図に示します。

図2.15 パイプライン化したPut通信



この例では以下の流れでPut通信の中継が行われます。図の括弧内の数字が以下の手順に該当します。

1. 中継プロセスBで、終端プロセスC宛ての3つのPutディスクリプタを書いておく。1つ目のPut通信は送受信バッファの先頭から10バイト、2つ目のPut通信は送受信バッファの11バイト目から10バイト、3つ目のPut通信は送受信バッファの21バイト目から10バイトとする。
2. 発端プロセスAで、SPSを1とした中継プロセスB宛ての3つのPut通信を指示する。それぞれのPut通信は1.と同じようにデータを分割する。
3. 2.により、発端プロセスAで2.のPut通信が順次開始される。
4. 3.の1つ目のPut通信が中継プロセスBに届くと、1.の1つ目のPut通信が自動的に開始される。
5. 3.の2つ目のPut通信が中継プロセスBに届くと、1.の2つ目のPut通信が自動的に開始される。
6. 3.の3つ目のPut通信が中継プロセスBに届くと、1.の3つ目のPut通信が自動的に開始される。

“2.2.10 通信の完了確認と順序保証”に示すように、パイプライン化する場合は、発端プロセスAで実行するすべてのPut通信でローカルVCQや通信経路座標を同じにする必要があります。そうしないと、Put通信が中継プロセスBに届く順序が発端プロセスAで指示した順序とは異なることがあり、発端プロセスAからの受信がまだ完了していないデータを終端プロセスCに送信してしまうことがあります。

この例では中継プロセスはBだけですが、プロセスCも中継プロセスとすることで、3段以上にパイプライン化することもできます。

2.2.9 パケットの最大転送サイズと送信間隔

Tofuインターコネクトでは、データをパケットと呼ぶ単位に分割して転送します。1つのパケットの最大転送サイズをMTU(maximum transfer unit)と呼びます。MTUより大きいサイズのデータを送受信する場合は、データをMTUごとに分割してパケットとして転送します。

複数の通信の間で通信経路の衝突が起きることが事前に分かっている場合、パケットの送出帯域を抑制することで輻輳による実効帯域の低下を緩和できる場合があります。PutとGetでは、TNIがパケットを送信した後に次のパケットをすぐに送信せずに間隔を空け、送出帯域を抑制できます。この間隔を送信間隔と呼びます。

2.2.10 通信の完了確認と順序保証

Put通信では、リモート計算ノードでTNIが主記憶へデータを書き込みます。Get通信では、ローカル計算ノードでTNIが主記憶へデータを書き込みます。このとき、書込み先の領域が複数のCPUキャッシュラインにまたがる場合は、そのCPUキャッシュライン間での主記憶への書込み順序は保証されず、入れ替わることがあります。また、書込み先の領域が複数のCPUキャッシュラインにまたがっていなくても、読込み元の領域が複数のページにまたがっている場合は、ページ境界前後の領域間での主記憶への書込み順序は保証されず、入れ替わることがあります。このため、1つの通信において、CPUのロード命令などによってある領域が書き換わっていることが確認できても、別の領域も書き換わっているとは判断できません。ただし、書込み先の1つのCPUキャッシュラインに着目した場合、それに対応する読込み元の領域が複数のページにまたがっていない限り、CPUキャッシュラインの粒度で書込みが行われることは保証されます。したがって、書込み先における1つのCPUキャッシュラインの範囲では、それに対応する読込み元の領域が複数のページにまたがっていない限り、CPUのロード命令などによってある領域が書き換わっていることが確認できたら、別の領域も書き換わっていると判断できます。このため、CPUキャッシュラインの境界で分けられた領域ごとにCPUのロード命令などによって書込みを確認することで、全体の書込みの完了確認が可能です。ARMW通信では、領域が複数のCPUキャッシュラインにまたがることがありません。

CPUキャッシュラインの境界で分けられた領域ごとにCPUのロード命令などによって書込みすることを強く推奨します。そうすることで、TCQとMRQによって1つの通信におけるデータの読込み・書込みの完了は、以下のように確認できます。

- Put通信のローカル計算ノードにおいて、主記憶からのデータ読込みがすべて完了したことは、TCQ通知により確認できる。
- Put通信のリモート計算ノードにおいて、主記憶へのデータ書込みがすべて完了したことは、リモートMRQ通知により確認できる。
- Get通信のリモート計算ノードにおいて、主記憶からのデータ読込みがすべて完了したことは、リモートMRQ通知により確認できる。
- Get通信のローカル計算ノードにおいて、主記憶へのデータ書込みがすべて完了したことは、ローカルMRQ通知により確認できる。
- ARMW通信のリモート計算ノードにおいて、主記憶へのデータ書込みがすべて完了したことは、リモートMRQ通知により確認できる。

同一のローカルVCQのTOQに複数の通信指示を出したときの順序は、Put、Get、およびARMWの組合せにかかわらず、以下のように保証されます。ストロングオーダ・フラグ設定時の動作仕様については、“6.1.3 本システムの動作仕様”を参照してください。

- ローカルおよびリモートの主記憶の読込みや書込みにおける順序は、デフォルトでは保証されない。これをある程度保証するため、ストロングオーダ機能が存在する。あるTOQディスクリプタでストロングオーダ・フラグを設定すると、リモートVCQと通信経路座標が一致している場合に限り、その通信による読込みは先行する通信による読込みが完了した後に開始されることが保証され、その通信による書込みは先行する通信による書込みが完了した後に開始されることも保証される。
- TCQディスクリプタによる完了通知は、TOQディスクリプタによる通信指示と同じ順番で通知されることが保証される。
- リモートMRQディスクリプタによる完了通知は、リモートVCQと通信経路座標が一致している場合に限り、TOQディスクリプタによる通信指示と同じ順番で通知されることが保証される。
- ローカルMRQディスクリプタによる完了通知は、リモートVCQと通信経路座標が一致している場合に限り、TOQディスクリプタによる通信指示と同じ順番で通知されることが保証される。

このTCQディスクリプタの性質を利用して、TCQディスクリプタによる完了通知がどのTOQディスクリプタによる通信に対応するものなのかを知ることができます。また、TOQディスクリプタの書込みによって通信を開始する関数には、コールバックデータを指定する引数が存在します。TCQディスクリプタによる通信完了を通知する関数には、対応するTOQディスクリプタの書込み時に指定したコールバックデータを返却するための引数が存在します。このコールバックデータを使用してTOQディスクリプタとTCQディスクリプタの対応を知ることもできます。

ローカルMRQディスクリプタによる完了通知では、それがどのTOQディスクリプタによる通信に対応するものなのかを簡単に知ることはできません。しかし、TOQディスクリプタの書込みによって通信を開始する関数には、EDATAという値を指定する引数が存在します。MRQディスクリプタによる通信完了を通知する関数には、対応するTOQディスクリプタの書込み時に指定したEDATAの値を含む構造体オブジェクトを返却するための引数が存在します。このEDATAを使用してTOQディスクリプタとMRQディスクリプタの対応を知ることができます。

この構造体オブジェクトはリモートMRQディスクリプタによる完了通知でも返却されます。そのため、EDATAを使えば、リモートプロセスにおいて何の通信が完了したのかを知ることもできます。

2.2.11 キャッシュインジェクションとパディング

Putにおけるリモート計算ノードでのTNIによる主記憶へのデータ書込みと、Getにおけるローカル計算ノードでのTNIによる主記憶へのデータ書込みにおいて、その後のCPUによる主記憶からのデータ読込みを高速にするため、キャッシュインジェクション機能があります。

TOQディスクリプタにキャッシュインジェクション・フラグを設定すると、主記憶だけでなくCPUの最終レベルキャッシュに対してもデータが書き込まれます。これによって、書き込まれたデータにCPUがアクセスするときに、最終レベルキャッシュでのキャッシュミスを抑止できます。

ただし、キャッシュインジェクションが機能するには条件があり、書込みがキャッシュラインを完全に上書きし、なおかつクリーンなキャッシュラインにヒットする必要があります。クリーンなキャッシュラインが最終レベルキャッシュに載っている典型的なケースは、CPUが該当キャッシュラインをポーリングしている場合です。この条件を満たさないキャッシュラインでは主記憶だけにデータが書き込まれ、その後のCPUによるデータ読込みではキャッシュミスが発生します。

ピギーバックを使用したPutでは、転送サイズがキャッシュラインのサイズに満たないため、キャッシュラインを完全に上書きできません。そのため、TOQディスクリプタにキャッシュインジェクション・フラグを設定しただけでは、CPUの最終レベルキャッシュに書込みを行えません。

ピギーバックを使用したPutでもキャッシュミスを抑止できるようにするため、キャッシュラインパディング機能があります。TOQディスクリプタにキャッシュラインパディング・フラグを設定すると、データの先頭および末尾をキャッシュライン境界まで不定値で拡張して書き込みます。したがって、キャッシュインジェクション・フラグとキャッシュラインパディング・フラグを同時に指定することにより、CPUの最終レベルキャッシュに書込みを行えます。キャッシュラインパディング・フラグはピギーバックを使用したPutだけで使用可能です。

2.2.12 通信エラー

ワンサイド通信では、ローカルTNI、リモートTNI、および通信経路上のTNR・Tofuネットワークに障害が発生しておらず通信可能な状態になっていれば、送信元TNIから宛先TNIまでのパケット転送が成功することが保証されます。

しかし、指定したSTADDの値が誤っているなどの理由で通信エラーが発生することがあります。ローカルTNIでパケットを送出するときに発生したエラーはTCQ、それ以降に発生したエラーはMRQに通知されます。エラーの内容はワンサイド通信完了確認関数の復帰値で確認できます。

2.3 バリア通信

バリア通信では、TNIが複数の計算ノード間でパケットやシグナルを順次送受信することで、それらの計算ノード間でバリア同期を行います。バリア同期と同時にリダクション演算も可能となっています。

2.3.1 VBG(Virtual Barrier Gate)

各TNIはBG(barrier gate)と呼ぶ回路を持ち、各BGがパケットやシグナルの送受信を行います。バリア通信を使用するには、バリア同期を開始する前に、バリア同期に参加するすべての計算ノードでBGの設定を済ませておく必要があります。バリア同期に先立って設定されるBG間通信経路のネットワークをバリア回路と呼びます。

あるバリア回路の構築のためにTNIあたり複数のBGを使用する場合、各TNIにおける複数のBG間には一続きで循環した依存関係が設定されます。BGのうち1つは始点・終点BGとなり、それ以外のBGは中継BGとなります。一連のBGによるパケット送信は始点・終点BGから開始され、始点・終点BGの手前のBGで終わります。パケットの送信順序を制御するため、各BGはパケットを送信すると同時に、次のBGにシグナルを送ります。

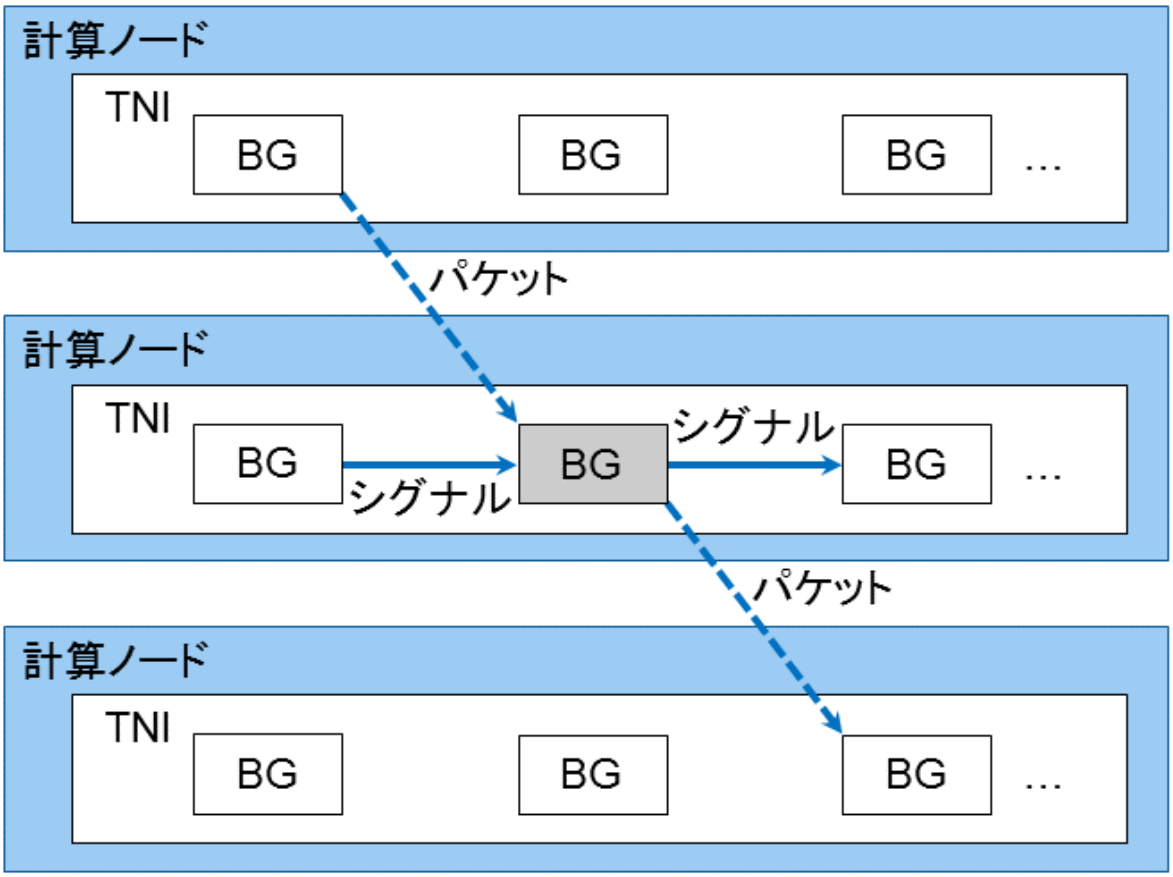
始点・終点BGは、バリア通信が開始されると、あらかじめ設定された通り、任意のTNIのBGにパケットを、同一TNI内の別のBGにシグナルを、送出します。その後、あらかじめ設定された通り、任意のTNIのBGからのパケットと、同一TNI内の別のBGからのシグナルを、待ち合わせます。

中継BGは、あらかじめ設定されたとおり、任意のTNIのBGからのパケットと、同一TNI内の別のBGからのシグナルを、待ち合わせます。その両方が届くと、あらかじめ設定されたとおり、任意のTNIのBGにパケットを、同一TNI内の別のBGにシグナルを、送出します。

パケットの送出元BGと送出先BGは、同一計算ノード内のTNIであっても、別の計算ノードのTNIであってもかまいません。

パケット・シグナルの送出を下図に示します。この図では、中央の1つのBGに着目し、そのBGが送出先・送出元となっているパケット・シグナルだけが描かれています。

図2.16 BGとパケット・シグナル



uTofuでは、BGをVBG(virtual barrier gate)として抽象化しています。それぞれのVBGにはVBG IDが与えられます。VBG IDは、VBGを特定する64ビット符号なし整数であり、Tofuネットワーク内のすべてのVBG間で一意の値を持ちます。

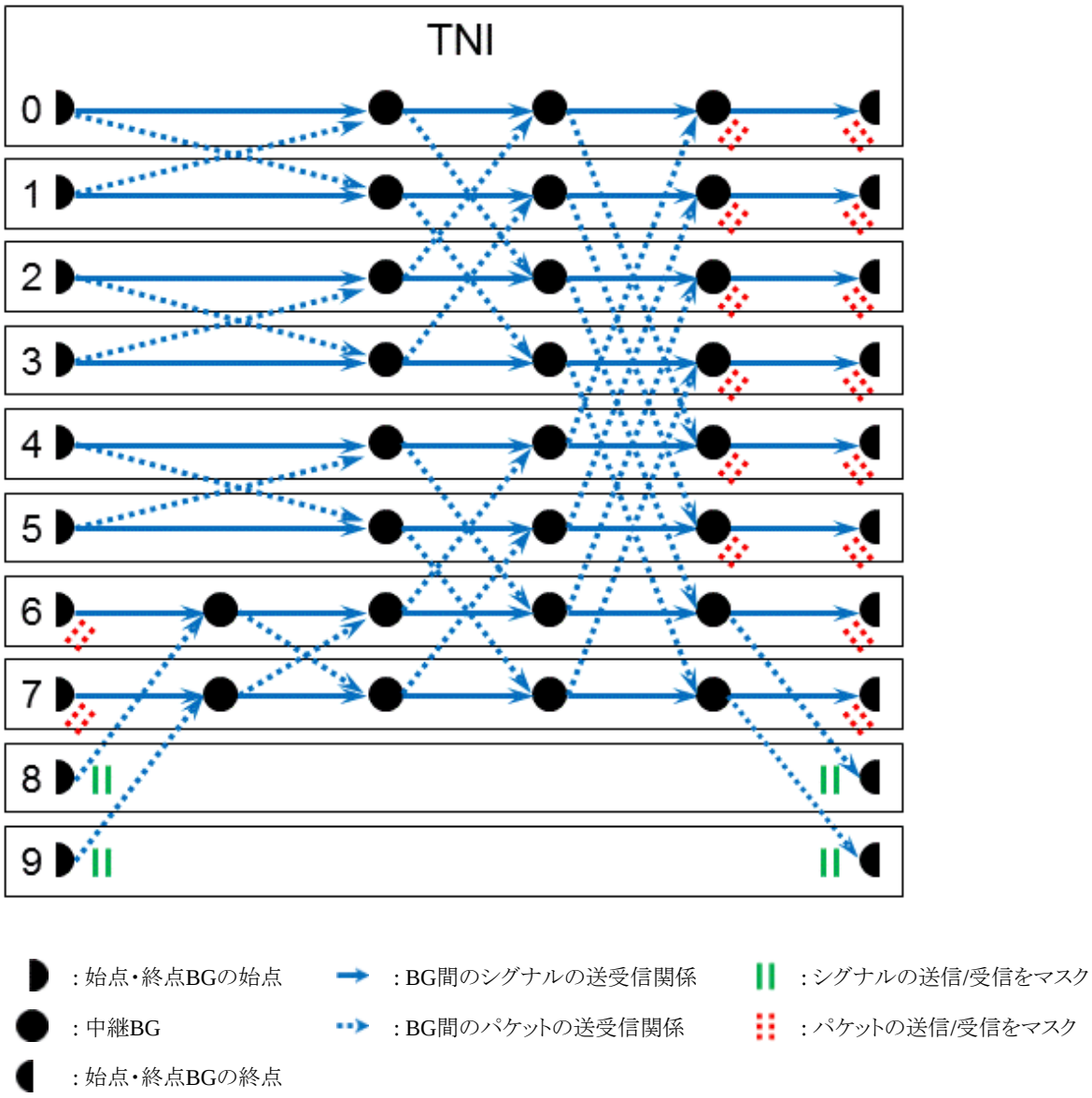
2.3.2 バリア回路

バリア同期に参加する各BGに以下の情報を設定することにより、バリア回路を構築できます。VBG IDは“3.4.1 VBG確保・解放関数”で説明されている関数で取得します。VBG IDの設定例は“5.2.1 バリア同期とリダクション演算の例”を参照してください。

- シグナルの送信元VBG ID (同一TNIの別のVBG)
- パケットの送信元VBG ID (任意のVBG)
- シグナルの送信先VBG ID (同一TNIの別のVBG)
- パケットの送信先VBG ID (任意のVBG)

10プロセスでバタフライ交換アルゴリズムによるバリア回路を設定する場合の例を下図に示します。

図2.17 10プロセスでのバタフライ交換アルゴリズムによるバリア回路



“3.4.1 VBG確保・解放関数”で説明されている関数で取得したVBGは、“3.4.2 VBG設定関数”で説明されている関数で設定します。

バリア通信を開始する前に、バリア同期に参加するすべてのVBGで設定を完了する必要があります。例えば、Tofuインターコネクトのバリア通信をMPIのバリア同期に使用する場合は、該当のコミュニケータを作成するタイミングでVBGの設定を行い、すべてのVBGの設定が完了したことをワンサイド通信などのなんらかの方法で確認するのが、典型的な使い方です。

2.3.3 バリア同期

バリア通信を開始するuTofuの関数は、TNIに対してバリア通信の開始を指示したタイミングで復帰します。バリア同期を確認するには、バリア通信の完了を確認するuTofuの関数を呼び、完了を示す復帰値が返却されるまでポーリングする必要があります。

使用するVBGが異なれば、複数のバリア通信を同時に実行できます。

バリア通信のプログラム例については、“5.2.1 バリア同期とリダクション演算の例”を参照してください。

2.3.4 リダクション演算

バリア通信の開始時に入力データを与えることで、バリア同期を行いながら同時にリダクション演算を行うことができます。

uTofuで使用可能な演算種別とその型を下表に示します。ただし、実際に使用可能な演算種別は、Tofuインターコネクトの種類によって異なります。

本システム固有の情報については、“第6章 システムの情報”を参照してください。

表2.3 リダクションの演算種別

名称	型	演算
BAND	64ビット符号なし整数	ビットごとの論理積
BOR	64ビット符号なし整数	ビットごとの論理和
BXOR	64ビット符号なし整数	ビットごとの排他的論理和
MAX	64ビット符号なし整数	最大値
MAXLOC	64ビット符号なし整数	(注)
SUM	64ビット符号なし整数	和
BFPSUM	倍精度浮動小数点	和

注) MAXLOCは、64ビット符号なし整数2組を入力とし、1番目の要素についてはMAXと同様に最大値を計算します。2番目の要素は、1番目の要素で最大値を格納していた入力と、同じ組の要素を選択します。ここで1番目の要素の値が等しい場合は、2番目の要素同士を比較して小さい方を選択します。

例えばSUMでは、各プロセスが64ビット符号なし整数を出し合い、その総和を求めることができます。

1回のバリア通信で実行可能なリダクション演算の要素数は、Tofuインターコネクトの種類によって異なります。複数の要素のリダクション演算が可能な場合は、配列の形でデータを与え、同じ位置の要素同士で演算を行います。ただし、4つ以上のリダクション演算が可能な場合のMAXLOCでは、要素を2組ずつに区切って、奇数番目の要素と偶数番目の要素で、前述の演算を行います。

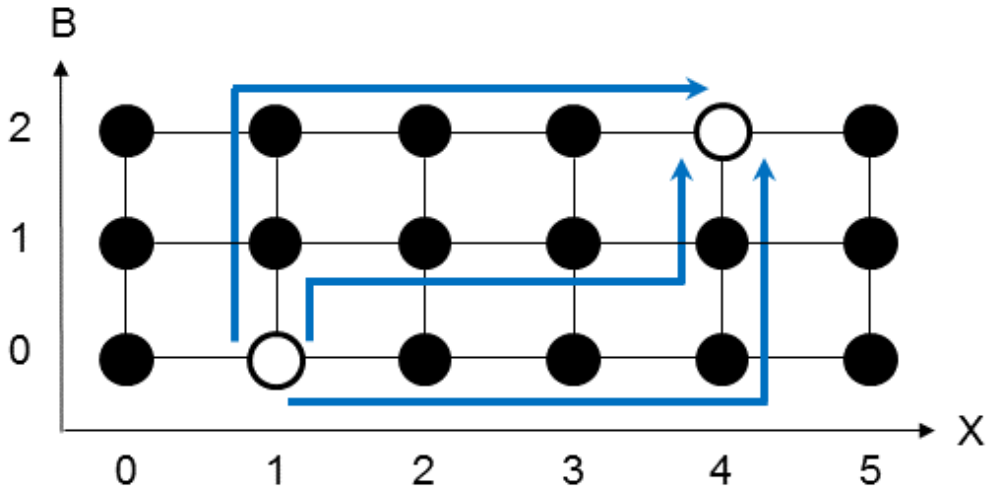
リダクション演算のプログラム例については、“5.2.1 バリア同期とリダクション演算の例”を参照してください。

2.4 通信経路

Tofuネットワークは6次元のメッシュ/トーラスのトポロジーを持つため、パケットがある計算ノードから別の計算ノードに到達するには複数の経路が存在します。その通信経路は通信経路座標と宛先計算ノード座標との組で決定されます。通信経路座標はA、B、C座標、宛先計算ノード座標はX、Y、Z、A、B、C座標で指定します。パケットはA、B、C軸を通信経路として指定された座標へ移動し、その後宛先計算ノードへ向かってX、Y、Z軸とA、B、C軸を移動します。例えば、送信元計算ノード座標が(X_S, Y_S, Z_S, A_S, B_S, C_S)で、宛先計算ノード座標を(X_D, Y_D, Z_D, A_D, B_D, C_D)、通信経路座標を(A_P, B_P, C_P)と指定すると、パケットはまず座標(X_S, Y_S, Z_S, A_P, B_P, C_P)に最短経路で移動し、その後(X_D, Y_D, Z_D, A_P, B_P, C_P)に最短経路で移動し、最後に座標(X_D, Y_D, Z_D, A_D, B_D, C_D)に最短経路で移動します。A、B、C軸はそれぞれ2、3、2の長さを持つため、通信経路は最大で12通り存在します。

X、Bの2次元面での通信経路の例を下図に示します。この例では、X、B座標が(1, 0)の計算ノードから(4, 2)の計算ノードへの通信において通信経路のB座標を(0)、(1)、(2)としたときの3つの通信経路が描かれています。

図2.18 X、Bの2次元面での3つの通信経路



ワンサイド通信では1つの通信でローカル計算ノードとリモート計算ノードの間でパケットが往復します。通信経路座標で指定するのはローカル計算ノードからリモート計算ノードへの通信経路です。リモート計算ノードからローカル計算ノードへの通信経路は、それと同じ通信経路を逆にたどります。

2.5 スレッド安全性

uTofuインターフェースでは、スレッド安全性に関する指定をしなくても、以下の場合を除けば、複数のスレッドから同時に呼び出したときの動作が保証されます。

- CQを共有する、スレッド安全ではない1つまたは複数のVCQに対して、以下に分類されるどれかの関数を同時に呼び出したとき
  - STADDの管理
  - ワンサイド通信の実行
- スレッド安全でない1つのVBGに対して、以下に分類されるどれかの関数を同時に呼び出したとき
  - バリア通信の実行

上記の関数も含めて複数のスレッドから同時に呼び出すために、VCQの作成時には、スレッド安全なCQ(スレッド安全VCQ)、または、CQを共有しないVCQ(CQ専有VCQ)を作成するように指示できます。また、VBGの作成時には、スレッド安全なVBG(スレッド安全VBG)を作成するように指示できます。ただし、スレッド安全VCQやスレッド安全VBGでは、関数の呼出し時のソフトウェアオーバーヘッドが増加する場合があります。

第3章 uTofuインターフェース仕様

この章では、uTofuのインターフェース仕様について説明します。

uTofuはC言語(C99(ISO/IEC 9899:1999規格)またはそれ以降)のインターフェースを提供します。uTofuを使用するプログラムは以下のようにヘッダファイルutofu.hを取り込む必要があります。

```
#include <utofu.h>
```

uTofuの機能は大きく以下に分類されます。また、機能に対応する関数の詳細については下表に示す参照先の説明をお読みください。

表3.1 uTofuの機能と関数

機能分類	機能	対応する関数の詳細
TNIの問合せ	TNI問合せ	“3.2.1 TNI問合せ関数”
VCQの管理	VCQ作成・解放	“3.3.1 VCQ作成・解放関数”
	VCQ ID操作	“3.3.2 VCQ ID操作関数”
	VCQ問合せ	“3.3.3 VCQ問合せ関数”
VBGの管理	VBG確保・解放	“3.4.1 VBG確保・解放関数”
	VBG設定	“3.4.2 VBG設定関数”
	VBG問合せ	“3.4.3 VBG問合せ関数”
通信経路の管理	通信経路管理	“3.5.1 通信経路管理関数”
STADDの管理	STADD管理	“3.6.1 STADD管理関数”
ワンサイド通信の実行	ワンサイド通信開始	“3.7.1 ワンサイド通信開始関数”
	ワンサイド通信準備	“3.7.2 ワンサイド通信準備関数”
	ワンサイド通信一括開始	“3.7.3 ワンサイド通信一括開始関数”
	ワンサイド通信完了確認	“3.7.4 ワンサイド通信完了確認関数”
	ワンサイド通信問合せ	“3.7.5 ワンサイド通信問合せ関数”
バリア通信の実行	バリア通信開始	“3.8.1 バリア通信開始関数”
	バリア通信完了確認	“3.8.2 バリア通信完了確認関数”
補助機能	バージョン情報問合せ	“3.9.1 バージョン情報問合せ関数”
	計算ノード情報問合せ	“3.9.2 計算ノード情報問合せ関数”

以降の節では、これらの機能ごとにインターフェース仕様を説明します。

なお、この章では下表に示す表記を使用します。

表3.2 インターフェース仕様の説明で使用する表記

表記	意味
a_*	a_ から始まる識別子を持つ、任意の関数、列挙定数、またはマクロ。
a_{b|c}	a_bまたはa_cという識別子を持つ、関数、列挙定数、またはマクロ。
a::b	aという構造体または共用体のbというメンバ、またはaという関数の仮引数b 。

表3.3 関数の引数の表のIN/OUT列で使用する表記

表記	意味
IN	関数への入力。
OUT	関数からの出力。
IN,OUT	関数への入力、かつ、関数からの出力。

3.1 共通の定義

3.1.1 型定義

3.1.1.1 typedef

IDなどの型定義。

説明

uTofuで扱うIDなどの変数には型定義された専用の型を使用します。

型定義

型定義名	型	説明
utofu_tni_id_t	uint16_t	TNI(Tofu network interface) ID。 TNI IDは16ビット符号なし整数であり、計算ノード内のTNIを特定します。値は0から“計算ノードの持つTNIの数 -1”までの範囲です。TNI IDは計算ノード内で一意の値です。
utofu_cq_id_t	uint16_t	CQ(control queue) ID。 CQ IDは16ビット符号なし整数であり、TNI内のCQを特定します。値は0から“TNIの持つCQ数 -1”までの範囲です。CQ IDはTNI内で一意の値です。
utofu_bg_id_t	uint16_t	BG(barrier gate) ID。 BG IDは16ビット符号なし整数であり、TNI内のBGを特定します。値は0から“TNIの持つBG数 -1”までの範囲です。BG IDはTNI内で一意の値です。
utofu_cmp_id_t	uint16_t	コンポーネントID。 コンポーネントIDは16ビット符号なし整数であり、CQを共有する複数のVCQの中でVCQを特定します。コンポーネントIDはCQ内で一意の値です。
utofu_vcq_hdl_t	uintptr_t	VCQ(virtual control queue)ハンドル。 VCQハンドルはポインタサイズの符号なし整数であり、プロセス内でVCQのデータを参照します。VCQハンドルはプロセス内で一意の値です。
utofu_vcq_id_t	uint64_t	VCQ(virtual control queue) ID。 VCQ IDは64ビット符号なし整数であり、Tofuネットワーク内のVCQを特定します。VCQ IDはTofuネットワーク内のすべてのVCQ内で一意の値です。VCQ IDにデフォルト通信経路座標を埋め込むことができます。
utofu_vbg_id_t	uint64_t	VBG(virtual barrier gate) ID。 VBG IDは64ビット符号なし整数であり、Tofuネットワーク内のVBGを特定します。VBG IDはTofuネットワーク内のすべてのVBG内で一意の値です。
utofu_path_id_t	uint8_t	通信経路ID。 通信経路IDは8ビット符号なし整数であり、Tofuネットワーク上でのある計算ノードから別の計算ノードへの通信経路を示します。通信経路はバリア通信でも使用されますが、通信経路IDはワンサイド通信だけで使用されます。
utofu_stadd_t	uint64_t	STADD(steering address)。 STADDは64ビット符号なし整数であり、TNIが理解できるメモリアドレスです。ワンサイド通信で使用されます。

3.1.2 復帰値

3.1.2.1 enum utofu_return_code

uTofu関数の復帰値。

説明

下表に示す列挙定数はuTofuのほぼすべての関数の復帰値として使われます。

処理の成功を示す列挙定数UTOFU_SUCCESSの値は0です。それ以外の列挙定数は負の値です。UTOFU_SUCCESS以外の列挙定数は必ずしもエラーを表すものではありません。例えば、utofu_poll_tcq()関数とutofu_poll_mrq()関数は新しいTCQ/MRQエントリが見つからない場合にUTOFU_ERR_NOT_FOUNDを返却しますが、これは正常な状況です。

本書の各関数の説明にある復帰値の一覧では、以下に説明する列挙定数のうち、UTOFU_ERR_INVALID_ARGより前に書かれた復帰値だけを個別に説明しています。関数に渡された実引数やそのときの状態によって、そのほかの復帰値が返却されることがあります。

列挙定数

列挙定数	説明
UTOFU_SUCCESS	何の問題もなく処理が成功しました。
UTOFU_ERR_NOT_FOUND	新しいエントリが見つかりませんでした。
UTOFU_ERR_NOT_COMPLETED	処理はまだ完了していません。
UTOFU_ERR_NOT_PROCESSED	処理は実行されませんでした。
UTOFU_ERR_BUSY	資源が現在稼働中でふさがっています。 あとで繰り返せば成功する可能性があります。
UTOFU_ERR_USED	資源はすでに使用されています。 別の資源を指定すれば成功する可能性があります。
UTOFU_ERR_FULL	資源は満杯です。 これ以上の資源を確保できません。
UTOFU_ERR_NOT_AVAILABLE	資源は利用できません。 その資源を確保できません。
UTOFU_ERR_NOT_SUPPORTED	処理はサポートされていません。
UTOFU_ERR_TCQ_OTHER	ほかのUTOFU_ERR_TCQ_*で表せないエラーがTCQディスクリプタによって報告されました。
UTOFU_ERR_TCQ_DESC	TOQディスクリプタのエラーがTCQディスクリプタによって報告されました。 誤った引数が指定された可能性があります。
UTOFU_ERR_TCQ_MEMORY	メモリアクセスのエラーがTCQディスクリプタによって報告されました。 指定されたSTADDが不正なものである可能性があります。
UTOFU_ERR_TCQ_STADD	STADDのエラーがTCQディスクリプタによって報告されました。 指定されたSTADDの値が登録されたメモリ領域の範囲外である可能性があります。
UTOFU_ERR_TCQ_LENGTH	“STADD + データの長さ”のエラーがTCQディスクリプタによって報告されました。 “指定されたSTADD + 指定されたデータの長さ”の値が登録されたメモリ領域の範囲外である可能性があります。
UTOFU_ERR_MRQ_OTHER	ほかのUTOFU_ERR_MRQ_*で表せないエラーがMRQディスクリプタによって報告されました。
UTOFU_ERR_MRQ_PEER	通信相手プロセスのエラーがMRQディスクリプタによって報告されました。 このエラーは通信相手プロセスの異常終了が原因の可能性があります。
UTOFU_ERR_MRQ_LCL_MEMORY	ローカルメモリアクセスのエラーがMRQディスクリプタによって報告されました。 指定されたローカルSTADDが不正なものである可能性があります。
UTOFU_ERR_MRQ_RMT_MEMORY	リモートメモリアクセスのエラーがMRQディスクリプタによって報告されました。 指定されたリモートSTADDが不正なものである可能性があります。
UTOFU_ERR_MRQ_LCL_STADD	ローカルSTADDのエラーがMRQディスクリプタによって報告されました。 指定されたローカルSTADDの値が登録されたメモリ領域の範囲外である可能性があります。

列挙定数	説明
UTOFU_ERR_MRQ_RMT_STADD	リモートSTADDのエラーがMRQディスクリプタによって報告されました。 指定されたリモートSTADDの値が登録されたメモリ領域の範囲外である可能性があります。
UTOFU_ERR_MRQ_LCL_LENGTH	“ローカルSTADD + データの長さ”のエラーがMRQディスクリプタによって報告されました。 “指定されたローカルSTADD + 指定されたデータの長さ”の値が登録されたメモリ領域の範囲外である可能性があります。
UTOFU_ERR_MRQ_RMT_LENGTH	“リモートSTADD + データの長さ”のエラーがMRQディスクリプタによって報告されました。 “指定されたリモートSTADD + 指定されたデータの長さ”の値が登録されたメモリ領域の範囲外である可能性があります。
UTOFU_ERR_BARRIER_OTHER	ほかのUTOFU_ERR_BARRIER_*で表せないバリア通信のエラーが発生しました。
UTOFU_ERR_BARRIER_MISMATCH	バリア通信でリダクション演算の不一致が検出されました。
UTOFU_ERR_INVALID_ARG	不正な引数が指定されました。
UTOFU_ERR_INVALID_POINTER	不正なポインタが引数に指定されました。
UTOFU_ERR_INVALID_FLAGS	不正なフラグが引数に指定されました。
UTOFU_ERR_INVALID_COORDS	不正な計算ノード座標が引数に指定されました。
UTOFU_ERR_INVALID_PATH	不正な通信経路座標が引数に指定されました。
UTOFU_ERR_INVALID_TNI_ID	不正なTNI IDが引数に指定されました。
UTOFU_ERR_INVALID_CQ_ID	不正なCQ IDが引数に指定されました。
UTOFU_ERR_INVALID_BG_ID	不正なBG IDが引数に指定されました。
UTOFU_ERR_INVALID_CMP_ID	不正なコンポーネントIDが引数に指定されました。
UTOFU_ERR_INVALID_VCQ_HDL	不正なVCQハンドルが引数に指定されました。
UTOFU_ERR_INVALID_VCQ_ID	不正なVCQ IDが引数に指定されました。
UTOFU_ERR_INVALID_VBG_ID	不正なVBG IDが引数に指定されました。
UTOFU_ERR_INVALID_PATH_ID	不正な通信経路IDが引数に指定されました。
UTOFU_ERR_INVALID_STADD	不正なSTADDが引数に指定されました。
UTOFU_ERR_INVALID_ADDRESS	不正なメモリアドレスが引数に指定されました。
UTOFU_ERR_INVALID_SIZE	不正なサイズ・長さが引数に指定されました。
UTOFU_ERR_INVALID_STAG	不正なSTagが引数に指定されました。
UTOFU_ERR_INVALID_EDATA	不正なEDATAが引数に指定されました。
UTOFU_ERR_INVALID_NUMBER	不正な数が引数に指定されました。
UTOFU_ERR_INVALID_OP	不正な演算が引数に指定されました。
UTOFU_ERR_INVALID_DESC	不正なディスクリプタが引数に指定されました。
UTOFU_ERR_INVALID_DATA	不正な構造体データが引数に指定されました。
UTOFU_ERR_OUT_OF_RESOURCE	資源(メモリ以外)が枯渇しました。
UTOFU_ERR_OUT_OF_MEMORY	メモリを確保できませんでした。
UTOFU_ERR_FATAL	致命的なエラーが発生しました。

3.2 TNIの問合せ

TNI(Tofu network interface)は、各計算ノードに1つ以上備えられた物理的なネットワークインターフェースであり、ワンサイド通信とバリア通信を実行します。

使用可能なTNIはutofu_get_onesided_tnis()関数とutofu_get_barrier_tnis()関数で問い合わせることができます。

使用可能なワンサイド通信とバリア通信の機能はutofu_query_onesided_caps()関数とutofu_query_barrier_caps()関数で問い合わせることができます。

3.2.1 TNI問合せ関数

3.2.1.1 utofu_get_onesided_tnis

ワンサイド通信に使用可能なTNIのIDの配列を取得します。

書式

```
int utofu_get_onesided_tnis(
    utofu_tni_id_t **tni_ids,
    size_t          *num_tnis)
```

説明

この関数は、ワンサイド通信を実行でき、このプロセスが使用可能な、ローカルTNIのIDを返却します。使用可能なTNIの数は計算ノードが備えるTNIの数より少ない場合があります。

取得したTNI IDはutofu_query_onesided_caps()関数やutofu_create_vcq()関数の実引数として使用できます。

返却されたtni_ids配列は、使用可能なTNIが存在しない場合を除き、関数呼出し元がfree()関数によって解放する必要があります。

引数

仮引数名	説明	IN/OUT
tni_ids	メモリ確保されたTNI IDの配列へのポインタ。 配列長はnum_tnisです。使用可能なTNIが存在しない場合はNULLが設定されます。	OUT
num_tnis	使用可能なTNIの数。 使用可能なTNIが存在しない場合は0が設定されます。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.2.1.2 utofu_get_barrier_tnis

バリア通信に使用可能なTNIのIDの配列を取得します。

書式

```
int utofu_get_barrier_tnis(
    utofu_tni_id_t **tni_ids,
    size_t          *num_tnis)
```

説明

この関数は、バリア通信を実行でき、このプロセスが使用可能な、ローカルTNIのIDを返却します。使用可能なTNIの数は計算ノードが備えるTNIの数より少ない場合があります。

取得したTNI IDはutofu_query_barrier_caps()関数やutofu_alloc_vbg()関数の実引数として使用できます。

返却されたtni_ids配列は、使用可能なTNIが存在しない場合を除き、関数呼出し元がfree()関数によって解放する必要があります。

引数

仮引数名	説明	IN/OUT
tni_ids	メモリ確保されたTNI IDの配列へのポインタ。 配列長はnum_tnisです。使用可能なTNIが存在しない場合はNULLが設定されます。	OUT

仮引数名	説明	IN/OUT
num_tnis	使用可能なTNIの数。 使用可能なTNIが存在しない場合は0が設定されます。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.2.1.3 utofu_query_onesided_caps

ワンサイド通信で使用可能な機能や制限値を問い合わせます。

書式

```
int utofu_query_onesided_caps(
    utofu_tni_id_t               tni_id,
    struct utofu_onesided_caps **tni_caps)
```

説明

この関数は、uTofu実装が管理するメモリのデータへのポインタを返却します。関数呼出し元はtni_capsのメモリを解放したり上書きしたりしてはいけません。

utofu_get_onesided_tnis()関数が返却したTNIだけについて問い合わせることができます。

引数

仮引数名	説明	IN/OUT
tni_id	TNI ID。	IN
tni_caps	ワンサイド通信に関するTNIの機能のデータへのポインタ。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.2.1.4 utofu_query_barrier_caps

バリア通信で使用可能な機能や制限値を問い合わせます。

書式

```
int utofu_query_barrier_caps(
    utofu_tni_id_t              tni_id,
    struct utofu_barrier_caps **tni_caps)
```

説明

この関数は、uTofu実装が管理するメモリのデータへのポインタを返却します。関数呼出し元はtni_capsのメモリを解放したり上書きしたりしてはいけません。

utofu_get_barrier_tnis()関数が返却したTNIだけについて問い合わせることができます。

引数

仮引数名	説明	IN/OUT
tni_id	TNI ID。	IN
tni_caps	バリア通信に関するTNIの機能のデータへのポインタ。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.2.2 使用可能な通信機能や制限値を示す構造体

3.2.2.1 struct utofu_onesided_caps

ワンサイド通信の機能。

定義

```
struct utofu_onesided_caps {
    unsigned long int flags;
    unsigned long int armw_ops;
    unsigned int      num_cmp_ids;
    unsigned int      num_reserved_stags;
    size_t            cache_line_size;
    size_t            stag_address_alignment;
    size_t            max_toq_desc_size;
    size_t            max_putget_size;
    size_t            max_piggyback_size;
    size_t            max_edata_size;
    size_t            max_mtu;
    size_t            max_gap;
}
```

説明

utofu_query_onesided_caps()関数がこの構造体へのポインタを返却します。

メンバ

メンバ名	説明
flags	使用可能なワンサイド通信機能のフラグ。 UTOFU_ONESIDED_CAP_FLAG_*のビットごとのORです。
armw_ops	サポートしているARMW演算種別。 UTOFU_ONESIDED_CAP_ARMW_OP_*のビットごとのORです。
num_cmp_ids	使用可能なコンポーネントIDの数。 utofu_create_vcq_with_cmp_id()関数で使用されます。
num_reserved_stags	予約されたSTagのVCQごとの数。 utofu_reg_mem_with_stag()関数で使用されます。
cache_line_size	CPUの最終レベルキャッシュのラインサイズ。 キャッシュインジェクション機能や通信の完了確認で使用されます。
stag_address_alignment	utofu_reg_mem_with_stag()関数によってSTagを割り当てられるメモリアドレスのバイト境界。
max_toq_desc_size	1つのTOQディスクリプタの最大バイトサイズ。 utofu_prepare_*()関数で使用されます。
max_putget_size	1つのPutまたはGetの最大バイトサイズ。 ワンサイド通信開始関数やワンサイド通信準備関数のlength仮引数に指定する値はこの値以下である必要があります。
max_piggyback_size	ピギーバックの最大バイトサイズ。

メンバ名	説明
	utofu_put_piggyback()関数のlength仮引数に指定する値はこの値以下である必要があります。
max_edata_size	使用可能なEDATAフィールドの最大バイトサイズ。
max_mtu	パケットのMTU(最大転送サイズ)の最大値。
max_gap	パケットの間に挿入する送信間隔の最大値。

3.2.2.2 struct utofu_barrier_caps

バリア通信の機能。

定義

```
struct utofu_barrier_caps {
    unsigned long int flags;
    unsigned long int reduce_ops;
    size_t            max_uint64_reduction;
    size_t            max_double_reduction;
}
```

説明

utofu_query_barrier_caps()関数がこの構造体へのポインタを返却します。

メンバ

メンバ名	説明
flags	使用可能なバリア通信機能のフラグ。 UTOFU_BARRIER_CAP_FLAG_*のビットごとのORです。
reduce_ops	サポートしているリダクション演算種別。 UTOFU_BARRIER_CAP_REDUCE_OP_*のビットごとのORです。
max_uint64_reduction	同時に実行可能な64ビット整数(uint64_t)のリダクション演算の最大数。
max_double_reduction	同時に実行可能な浮動小数点(double)のリダクション演算の最大数。

3.2.3 使用可能な通信機能を示すフラグ

3.2.3.1 UTOFU_ONESIDED_CAP_FLAG_*

utofu_onesided_caps構造体のflagsメンバのビットフラグ。

説明

TNIがワンサイド通信のどの機能を持っているかを確認できます。

マクロ

マクロ名	説明
UTOFU_ONESIDED_CAP_FLAG_SESSION_MODE	セッションモード機能。
UTOFU_ONESIDED_CAP_FLAG_ARMW	Atomic Read Modify Write通信。

3.2.3.2 UTOFU_BARRIER_CAP_FLAG_*

utofu_barrier_caps構造体のflagsメンバのビットフラグ。

説明

TNIがバリア通信のどの機能を持っているかを確認できます。

現在のバージョンではフラグは定義されていません。

3.2.3.3 UTOFU_ONESIDED_CAP_ARMW_OP_*

utofu_onesided_caps構造体のarmw_opsメンバのビットフラグ。

説明

TNIがワンサイド通信のARMW(Atomic Read Modify Write)通信のどのARMW演算を使用可能かを確認できます。

マクロ

マクロ名	説明
UTOFU_ONESIDED_CAP_ARMW_OP_CSWAP	コンペア・アンド・スワップ。
UTOFU_ONESIDED_CAP_ARMW_OP_SWAP	スワップ。
UTOFU_ONESIDED_CAP_ARMW_OP_ADD	符号なし整数として加算。
UTOFU_ONESIDED_CAP_ARMW_OP_XOR	ビットごとの排他的論理和。
UTOFU_ONESIDED_CAP_ARMW_OP_AND	ビットごとの論理積。
UTOFU_ONESIDED_CAP_ARMW_OP_OR	ビットごとの論理和。

3.2.3.4 UTOFU_BARRIER_CAP_REDUCE_OP_*

utofu_barrier_caps構造体のreduce_opsメンバのビットフラグ。

説明

TNIがバリア通信のどのリダクション演算を使用可能かを確認できます。

マクロ

マクロ名	説明
UTOFU_BARRIER_CAP_REDUCE_OP_BARRIER	バリア同期(リダクション演算なし)。
UTOFU_BARRIER_CAP_REDUCE_OP_BAND	uint64_t型の値のビットごとの論理積。
UTOFU_BARRIER_CAP_REDUCE_OP_BOR	uint64_t型の値のビットごとの論理和。
UTOFU_BARRIER_CAP_REDUCE_OP_BXOR	uint64_t型の値のビットごとの排他的論理和。
UTOFU_BARRIER_CAP_REDUCE_OP_MAX	uint64_t型の値の符号なし整数としての最大値。
UTOFU_BARRIER_CAP_REDUCE_OP_MAXLOC	uint64_t型の値の符号なし整数としての最大値とその位置。
UTOFU_BARRIER_CAP_REDUCE_OP_SUM	uint64_t型の値の符号なし整数としての和。
UTOFU_BARRIER_CAP_REDUCE_OP_BFPSUM	double型の値の浮動小数点としての和。

3.3 VCQの管理

VCQ(virtual control queue)はuTofu使用者がTNIを使用してワンサイド通信を実行するためのインターフェースです。

ワンサイド通信を実行する前に、uTofu使用者は、utofu_get_onesided_tnis()関数を使用して使用可能なローカルTNIを問い合わせ、utofu_create_vcq()関数を使用してそのローカルTNIに対応したVCQを作成する必要があります。ワンサイド通信の使用が終了したら、uTofu使用者は、utofu_free_vcq()関数を使用してVCQを解放する必要があります。2つのプロセス間で通信を行う場合は、双方のプロセスでVCQを作成する必要があります。

ワンサイド通信の実行時には、通信相手となるリモートVCQをVCQ IDで指定します。ローカルVCQ IDはutofu_query_vcq_id()関数を使用して取得できます。ほかのプロセスと通信を行うには、そのプロセスからあらかじめVCQ IDを通知してもらう必要があります。ワンサイド通信では通信経路を指定する必要がありますが、utofu_set_vcq_id_path()関数を使用してデフォルト通信経路座標をVCQ IDに埋め込んでおくことができます。

作成したVCQの情報はutofu_query_vcq_info()関数を使用して取得できます。

1つのTNIに複数のVCQを作成することもできます。それぞれのVCQは独立した通信コンテキストを持ちます。

このプロセスでVCQを1つ作成したら、そのVCQはほかのプロセスのVCQと何度でも通信できます。

3.3.1 VCQ作成・解放関数

3.3.1.1 utofu_create_vcq

この計算ノードのTNIにVCQを作成します。

書式

```
int utofu_create_vcq(
    utofu_tni_id_t     tni_id,
    unsigned long int  flags,
    utofu_vcq_hdl_t   *vcq_hdl)
```

説明

作成したVCQはutofu_free_vcq()関数で解放する必要があります。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
tni_id	VCQを作成するTNIのID。	IN
flags	UTOFU_VCQ_FLAG_*のビットごとのOR。	IN
vcq_hdl	作成されたVCQのハンドル。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_FULL	このTNIにこれ以上VCQを作成できません。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のVCQを作成できません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のVCQはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.1.2 utofu_create_vcq_with_cmp_id

コンポーネントIDを指定して、この計算ノードのTNIにVCQを作成します。

書式

```
int utofu_create_vcq_with_cmp_id(
    utofu_tni_id_t     tni_id,
    utofu_cmp_id_t     cmp_id,
    unsigned long int  flags,
    utofu_vcq_hdl_t   *vcq_hdl)
```

説明

utofu_create_vcq()関数との違いは、上位コンポーネントのコンポーネントIDを指定できることです。コンポーネントIDは、CQを共有するほかの上位コンポーネントとVCQを区別するために使われます。utofu_create_vcq()関数を使うと、ほかのVCQと重複しないコンポーネントIDが選択され、それが作成されたVCQに割り当てられます。しかし、この関数を使う場合は、指定したコンポーネントIDがほかの上位コンポーネントに使われておらず使われることもないことを、関数呼出し元が保証する必要があります。コンポーネントIDが使われている場合は、この関数はUTOFU_ERR_USEDを返却します。

使用可能なコンポーネントIDは、0から“utofu_onesided_caps::num_cmp_ids -1”です。

作成したVCQはutofu_free_vcq()関数で解放する必要があります。

この関数はシステムコールの呼出しを伴うことがあります。

本書でのutofu_create_vcq()関数についての言及はこの関数にも適用されます。

引数

仮引数名	説明	IN/OUT
tni_id	VCQを作成するTNIのID。	IN
cmp_id	上位コンポーネントのID。 utofu_onesided_caps::num_cmp_ids未満の値を指定できます。	IN
flags	UTOFU_VCQ_FLAG_*のビットごとのOR。	IN
vcq_hdl	作成されたVCQのハンドル。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_USED	指定されたコンポーネントIDはほかのコンポーネントに使用されています。
UTOFU_ERR_FULL	このTNIにこれ以上VCQを作成できません。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のVCQを作成できません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のVCQはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.1.3 utofu_free_vcq

VCQを解放します。

書式

```
int utofu_free_vcq(
    utofu_vcq_hdl_t vcq_hdl)
```

説明

二重解放をしてはいけません。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vcq_hdl	utofu_create_vcq()関数によって作成されたVCQのハンドル。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.2 VCQ ID操作関数

3.3.2.1 utofu_query_vcq_id

ローカルVCQのIDを問い合わせます。

書式

```
int utofu_query_vcq_id(
    utofu_vcq_hdl_t  vcq_hdl,
    utofu_vcq_id_t  *vcq_id)
```

説明

VCQハンドルの値はこのプロセスだけで有効ですが、VCQ IDはすべてのプロセスで有効でありTofuネットワーク内のすべてのVCQで一意です。

デフォルト通信経路座標として、この計算ノードのA、B、C座標がVCQ IDに埋め込まれます。

引数

仮引数名	説明	IN/OUT
vcq_hdl	utofu_create_vcq()関数によって作成されたVCQのハンドル。	IN
vcq_id	VCQのID。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.2.2 utofu_construct_vcq_id

VCQ IDを生成します。

書式

```
int utofu_construct_vcq_id(
    uint8_t         coords[],
    utofu_tni_id_t  tni_id,
    utofu_cq_id_t   cq_id,
    utofu_cmp_id_t  cmp_id,
    utofu_vcq_id_t *vcq_id)
```

説明

VCQがutofu_create_vcq_with_cmp_id()関数で作成されたものである場合は、この関数を使用して、計算ノード座標、TNI ID、CQ ID、およびコンポーネントIDからそのVCQ IDを計算できます。ローカル計算ノードだけでなくリモート計算ノードで作成されたVCQのIDも計算することができます。

デフォルト通信経路座標として、指定された計算ノード座標のA、B、C座標がVCQ IDに埋め込まれます。

引数

仮引数名	説明	IN/OUT
coords	VCQの計算ノード座標(X、Y、Z、A、B、Cの順)。 配列長は6である必要があります。	IN
tni_id	VCQのTNI ID。	IN
cq_id	VCQのCQ ID。	IN
cmp_id	VCQのコンポーネントID。	IN
vcq_id	生成されたVCQ ID。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.2.3 utofu_set_vcq_id_path

VCQ IDに埋め込まれたデフォルト通信経路座標を更新します。

書式

```
int utofu_set_vcq_id_path(
    utofu_vcq_id_t *vcq_id,
    uint8_t         path_coords[])
```

説明

ワンサイド通信を呼び出すときに、リモートVCQ ID(rmt_vcq_id仮引数)またはビットフラグ(flags仮引数)によって、通信経路を指定できます。utofu_query_vcq_id()関数やutofu_construct_vcq_id()関数では、そのVCQに対応する計算ノードのA、B、C座標がデフォルト通信経路座標としてVCQ IDに埋め込まれます。この関数は、そのデフォルト通信経路座標を更新します。

この関数で更新される通信経路座標は局所的であり、uTofu実装が管理している情報が更新されるわけではないことにご注意ください。例えば、同じVCQ IDを持つ2つの変数があり、片方に対してこの関数を呼んだとしても、他方の通信経路座標は更新されません。

あるVCQ IDの変数に対してこの関数を複数回呼ぶこともできます。そのVCQ IDの変数を使用して通信する場合の通信経路座標は、この関数を最後に呼び出したときに設定した値になります。

引数

仮引数名	説明	IN/OUT
vcq_id	VCQ ID。	IN,OUT
path_coords	VCQに埋め込むデフォルト通信経路座標(A、B、Cの順)。 配列長は3である必要があります。NULLを指定することも可能であり、その場合は、ローカル計算ノードから指定されたVCQ IDの計算ノードまでの適切な通信経路を自動的に選択します。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.3 VCQ問合せ関数

3.3.3.1 utofu_query_vcq_info

VCQの計算ノード座標、TNI ID、およびCQ IDを問い合わせます。

書式

```
int utofu_query_vcq_info(
    utofu_vcq_id_t  vcq_id,
    uint8_t         coords[],
    utofu_tni_id_t *tni_id,
    utofu_cq_id_t  *cq_id,
    uint16_t       *extra_val)
```

説明

ローカル計算ノードだけでなくリモート計算ノードで作成されたVCQの情報も問い合わせることができます。

これらの情報は性能チューニングやデバッグに利用できる場合があります。例えば、1つの計算ノードに2つ以上のプロセスが存在し、それぞれのプロセスに別のTNIを使わせたい場合に、これらの情報を利用できます。

引数

仮引数名	説明	IN/OUT
vcq_id	VCQ ID。	IN
coords	VCQの計算ノード座標(X、Y、Z、A、B、Cの順)。 配列長は6以上である必要があります。	OUT
tni_id	VCQのTNI ID。	OUT

仮引数名	説明	IN/OUT
	TNI IDは0から“計算ノードの持つTNIの数 -1”までの範囲です。	
cq_id	VCQのCQ ID。 CQ IDは0から“TNIの持つCQの数 -1”までの範囲です。	OUT
extra_val	uTofu実装で内部的に使用される追加データ。 この値はuTofu使用者には意味を持ちません。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.3.4 VCQのフラグ

3.3.4.1 UTOFU_VCQ_FLAG_*

utofu_create_vcq()関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、作成するVCQの動作を指定できます。

マクロ

マクロ名	説明
UTOFU_VCQ_FLAG_THREAD_SAFE	スレッド安全VCQ。 このフラグは、スレッド安全なVCQを作成することをutofu_create_vcq()関数やutofu_create_vcq_with_cmp_id()関数に指示します。STADDの管理やワンサイド通信の実行を行う関数を、このフラグを指定して作成した1つのVCQに対して、複数のスレッドから同時に呼び出すことができます。また、それらの関数を、このフラグを指定して作成したVCQと別のVCQに対して、複数のスレッドから同時に呼び出すこともできます。
UTOFU_VCQ_FLAG_EXCLUSIVE	CQ専有VCQ。 このフラグは、ほかのVCQとCQを共有しないVCQを作成することをutofu_create_vcq()関数に指示します。STADDの管理やワンサイド通信の実行を行う関数を、このフラグを指定して作成したVCQと別のVCQに対して、複数のスレッドから同時に呼び出すことができます。しかし、このフラグは指定したがUTOFU_VCQ_FLAG_THREAD_SAFEフラグは指定せずに作成した1つのVCQに対して、それらの関数を複数のスレッドから同時に呼び出すことはできません。UTOFU_VCQ_FLAG_THREAD_SAFEフラグに対するこのフラグの利点は、異なるCQに対してそれらの関数を同時に呼び出した場合に、uTofu実装やハードウェアアクセスが並列に処理されることです。そのため、通信スループットが向上することがあります。CQが不足している場合には、このフラグが指定するとutofu_create_vcq()関数はエラーを返却します。utofu_create_vcq_with_cmp_id()関数にはこのフラグを使用できません。
UTOFU_VCQ_FLAG_SESSION_MODE	セッションモードVCQ。 このフラグは、セッションモードのVCQを作成することをutofu_create_vcq() 関数に指示します。TNIがセッションモードをサポートしていない場合や、セッションモード用のCQが不足している場合には、このフラグが指定するとutofu_create_vcq()関数はエラーを返却します。utofu_create_vcq_with_cmp_id()関数にはこのフラグを使用できません。このフラグとUTOFU_VCQ_FLAG_EXCLUSIVEフラグを同時には指定できません。

3.4 VBGの管理

VBG(virtual barrier gate)はTNIの構成要素であり、バリアシグナルやバリアパケットを送受信します。

バリア通信を開始する前に、uTofu使用者は、utofu_get_barrier_tnis()関数を使用して使用可能なTNIを問い合わせ、utofu_alloc_vbg()関数を使用してそのTNIからVBGを確保し、utofu_set_vbg()関数を使用してバリア回路を構築する必要があります。バリア通信の使用が終了したら、uTofu使用者は、utofu_free_vbg()関数を使用してVBGを解放する必要があります。確保したVBGの情報はutofu_query_vbg_info()関数を使用して取得できます。

リモート計算ノードと通信を行うためには、リモート計算ノードもVBGを確保する必要があります。

1つのTNIから複数のVBGを確保できます。むしろ、通常は、バリア回路を構築するためには複数のVBGが必要になります。

一度バリア回路を構築したら、そのバリア回路を使用してバリア同期やリダクション演算を何度でも実行できます。

3.4.1 VBG確保・解放関数

3.4.1.1 utofu_alloc_vbg

この計算ノードのTNIからVBGを確保します。

書式

```
int utofu_alloc_vbg(
    utofu_tni_id_t    tni_id,
    size_t            num_vbgs,
    unsigned long int flags,
    utofu_vbg_id_t    vbg_ids[])
```

説明

この関数は、関数呼出し元が用意したvbg_ids配列にVBG IDを書き込みます。

vbg_ids配列の最初の要素が始点・終点BGに、それ以降の要素が中継BGに対応します。

確保したVBGはutofu_free_vbg()関数で解放する必要があります。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
tni_id	BGを確保するTNIのID。	IN
num_vbgs	必要とするVBGの数。 この値は1以上である必要があります。	IN
flags	UTOFU_VBG_FLAG_*のビットごとのOR。	IN
vbg_ids	確保されたVBGのIDの配列。 配列長はnum_vbgs以上である必要があります。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_FULL	このTNIから指定された数のVBGを確保できません。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のVBGを確保できません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のVBGはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.4.1.2 utofu_free_vbg

VBGを解放します。

書式

```
int utofu_free_vbg(
    utofu_vbg_id_t vbg_ids[],
    size_t         num_vbgs)
```

説明

VBG IDはTofuネットワーク内のすべてのVBGで一意ですが、この関数ではそのプロセスが確保したVBGだけを解放できます。

二重解放をしてはいけません。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vbg_ids	utofu_alloc_vbg()関数によって確保されたVBGのIDの配列。 配列の内容はutofu_alloc_vbg()関数が返却した値と同じである必要があります。配列長はnum_vbgsである必要があります。	IN
num_vbgs	VBGの数。 この値は対応するutofu_alloc_vbg()関数の呼出しに使用したのと同じ値である必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.4.2 VBG設定関数

3.4.2.1 utofu_set_vbg

バリア回路を構築するためにVBGを設定します。

書式

```
int utofu_set_vbg(
    struct utofu_vbg_setting vbg_settings[],
    size_t                   num_vbg_settings)
```

説明

それぞれのVBGは、入力シグナルと入力パケットを受信し、出力シグナルと出力パケットを送信します。

- 入力シグナルは同一のTNIのほかのVBG(ローカル送信元)から受信します。
- 入力パケットは任意の計算ノードのTNIのVBG(リモート送信元)から受信します。
- 出力シグナルは同一のTNIのほかのVBG(ローカル送信先)に送信します。
- 出力パケットは任意の計算ノードのTNIのVBG(リモート送信先)に送信します。

バリア回路に参加するすべてのVBGを組み合わせることによってバリア回路が構築されます。1つのバリア回路を構築するには、そのバリア回路に参加するすべての計算ノードにおいて、それぞれのローカルVBG配列に対してこの関数を呼ぶ必要があります。

この関数は、1回のutofu_alloc_vbg()関数呼出しで確保した複数のVBGを、まとめて設定します。vbg_settings配列のutofu_vbg_setting::vbg_idの値の集合は、utofu_alloc_vbg()関数の返却したVBG IDの集合と同じ、またはその部分集合である必要があります。始点・終点BGに対応する、vbg_settings配列の最初のVBG IDは、utofu_alloc_vbg()関数が返却したVBG

IDの集合の最初のVBG IDである必要があります。しかし、残りのVBG IDは、utofu_alloc_vbg()関数の返却したVBG IDと同じ順番である必要はありません。

あるVBG IDに対してこの関数が複数回呼ばれた場合は、最後の呼出しだけが効果を持ちます。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vbg_settings	VBGの設定の配列。 配列長はnum_vbg_settingsである必要があります。	IN
num_vbg_settings	設定するVBGの数。 この値は、1以上、かつ、対応するutofu_alloc_vbg()関数の呼出しに使用した値以下である必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.4.3 VBG問合せ関数

3.4.3.1 utofu_query_vbg_info

VBGの計算ノード座標、TNI ID、BG IDを問い合わせます。

書式

```
int utofu_query_vbg_info(
    utofu_vbg_id_t  vbg_id,
    uint8_t         coords[],
    utofu_tni_id_t *tni_id,
    utofu_bg_id_t  *bg_id,
    uint16_t       *extra_val)
```

説明

ローカル計算ノードだけでなくリモート計算ノードで確保されたVBGの情報も問い合わせることができます。

これらの情報はデバッグに利用できる場合があります。

引数

仮引数名	説明	IN/OUT
vbg_id	VBG ID。	IN
coords	VBGの計算ノード座標(X、Y、Z、A、B、Cの順)。 配列長は6以上である必要があります。	OUT
tni_id	VBGのTNI ID。 TNI IDは0から“計算ノードの持つTNIの数 -1”までの範囲です。	OUT
bg_id	VBGのBG ID。 BG IDは0から“TNIの持つBGの数 -1”までの範囲です。	OUT
extra_val	uTofu実装で内部的に使用される追加データ。 この値はuTofu使用者には意味を持ちません。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.4.4 VBGの設定のための構造体

3.4.4.1 struct utofu_vbg_setting

VBGの設定。

定義

```
struct utofu_vbg_setting {
    utofu_vbg_id_t vbg_id;
    utofu_vbg_id_t src_lcl_vbg_id;
    utofu_vbg_id_t src_rmt_vbg_id;
    utofu_vbg_id_t dst_lcl_vbg_id;
    utofu_vbg_id_t dst_rmt_vbg_id;
    uint8_t        dst_path_coords[3];
}
```

説明

utofu_set_vbg()関数がvbg_settings仮引数にこの構造体の配列を取ります。

VBGがシグナル・パケットを受信・送信しない場合は、対応する送信元・送信先のローカル・リモートのVBG IDとしてUTOFU_VBG_ID_NULLを指定します。

メンバ

メンバ名	説明
vbg_id	設定するローカルVBGのID。
src_lcl_vbg_id	このVBGへの入力シグナルの送信元となるローカルVBGのID。
src_rmt_vbg_id	このVBGへの入力パケットの送信元となるリモートVBGのID。
dst_lcl_vbg_id	このVBGからの出力シグナルの送信先となるローカルVBGのID。
dst_rmt_vbg_id	このVBGからの出力パケットの送信先となるリモートVBGのID。
dst_path_coords	送信先となるリモートVBGへの通信経路座標(A、B、Cの順)。 座標A(dst_path_coords[0])にUTOFU_PATH_COORD_NULLを指定した場合は、適切な通信経路を自動的に選択します。その場合は、座標B(dst_path_coords[1])と座標C(dst_path_coords[2])は無視されます。

3.4.5 VBGのフラグ

3.4.5.1 UTOFU_VBG_FLAG_*

utofu_alloc_vbg()関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、確保するVBGの動作を指定できます。

マクロ

マクロ名	説明
UTOFU_VBG_FLAG_THREAD_SAFE	スレッド安全VBG。

マクロ名	説明
	このフラグは、スレッド安全なVBGを確保することをutofu_alloc_vbg()関数に指示します。バリア通信の実行を行う関数を、このフラグを指定して作成した1つのVBGに対して、複数のスレッドから同時に呼び出すことができます。

3.4.6 VBGの設定のための特別な値

3.4.6.1 UTOFU_VBG_ID_NULL

マクロ

マクロ名	説明
UTOFU_VBG_ID_NULL	無効なVBG ID。 utofu_set_vbg()関数に対して入力シグナル・パケットや出力シグナル・パケットが存在しないことを示すために使用する特別なVBG IDです。

3.5 通信経路の管理

ワンサイド通信を実行するときやバリア回路を構築するときに、Tofuネットワーク上の通信経路を指定する必要があります。

ワンサイド通信の場合は、ワンサイド通信開始関数やワンサイド通信準備関数の呼出し時に、リモートVCQ ID(rmt_vcq_id仮引数)またはビットフラグ(flags仮引数)によって、通信経路座標を指定できます。リモートVCQ IDでは、utofu_query_vcq_id()関数やutofu_construct_vcq_id()関数によってデフォルト通信経路座標がVCQ IDに埋め込まれ、utofu_set_vcq_id_path()関数によってそのVCQ IDに埋め込まれたデフォルト通信経路座標を更新できます。ビットフラグでは、UTOFU_ONESIDED_FLAG_PATH関数形式マクロによって通信経路を指定できます。UTOFU_ONESIDED_FLAG_PATH関数形式マクロは通信経路IDを引数に取ります。

通信経路IDはutofu_get_path_id()関数を使用して通信経路座標から作成できます。通信経路座標はutofu_get_path_coords()関数を使用して通信経路IDから問い合わせることができます。

バリア通信の場合は、VBG設定関数の呼出し時に通信経路座標を指定できます。

3.5.1 通信経路管理関数

3.5.1.1 utofu_get_path_id

通信経路座標から通信経路IDを作成します。

書式

```
int utofu_get_path_id(
    utofu_vcq_id_t   vcq_id,
    uint8_t          path_coords[],
    utofu_path_id_t *path_id)
```

説明

通信経路IDはリモートVCQ IDごとに定義します。したがって、通信経路座標が同じであっても、通信経路IDの作成時にこの関数に指定したリモートVCQ IDとは異なるリモートVCQ IDに対して、その通信経路IDを使用することはできません。

utofu_set_vcq_id_path()関数によってデフォルト通信経路座標を埋め込んだVCQ IDをvcq_id仮引数に指定することもできます。しかし、返却される通信経路IDはそのVCQ IDだけに有効な値です。したがって、VCQ IDに新しいデフォルト通信経路座標を埋め込んだ場合は、その新しいVCQ IDに対して新しい通信経路座標を生成するべきです。

引数

仮引数名	説明	IN/OUT
vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
path_coords	通信経路座標(A、B、Cの順)。	IN

仮引数名	説明	IN/OUT
	配列長は3である必要があります。NULLを指定することも可能であり、その場合は、ローカル計算ノードから指定されたVCQ IDの計算ノードまでの適切な通信経路を自動的に選択します。	
path_id	対応する通信経路ID。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.5.1.2 utofu_get_path_coords

通信経路IDから通信経路座標を問い合わせます。

書式

```
int utofu_get_path_coords(
    utofu_vcq_id_t  vcq_id,
    utofu_path_id_t path_id,
    uint8_t         path_coords[])
```

説明

通信経路IDはリモートVCQ IDごとに定義します。したがって、この関数に指定するリモートVCQ IDはutofu_get_path_id()関数に指定したリモートVCQ IDと一致している必要があります。

引数

仮引数名	説明	IN/OUT
vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
path_id	utofu_get_path_id()関数によって作成された通信経路ID。	IN
path_coords	対応する通信経路座標(A、B、Cの順)。 配列長は3以上である必要があります。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.5.2 通信経路の設定のための特別な値

3.5.2.1 UTOFU_PATH_COORD_NULL

マクロ

マクロ名	説明
UTOFU_PATH_COORD_NULL	無効な通信経路座標。 utofu_set_vbg()関数に対して適切な通信経路座標を自動的に選択することを示すために使用する、特別な通信経路座標です。

3.6 STADDの管理

STADD(steering address)はTNIが理解できるメモリアドレスです。ユーザー空間のプロセスで扱われるメモリアドレスはOSの割り当てる仮想アドレスであるため、TNIはそれを理解できません。ワンサイド通信においてTNIに適切なメモリアドレスを示すためにSTADDは使われます。

utofu_reg_mem()関数を使用してメモリ領域をVCQに登録することによって、VCQごとにSTADDが割り当てられます。割り当てられたSTADDはワンサイド通信開始関数やワンサイド通信準備関数に指定できます。割り当てられたSTADDは使用後にutofu_dereg_mem()関数を使用して解放しメモリ領域の登録も解除する必要があります。

utofu_reg_mem()関数を使用して、1つのメモリ領域を2回以上登録することができます。その場合、毎回同じSTADD値が返却されます。また、重なりのある複数のメモリ領域を登録することもできます。どちらの場合も、utofu_reg_mem()関数と同じ回数だけutofu_dereg_mem()関数を呼ぶ必要があります。

登録するメモリ領域は呼出しプロセスからアクセス可能でなければなりません。UTOFU_REG_MEM_FLAG_READ_ONLYフラグが指定されていない場合は、そのメモリ領域は読込み可能かつ書込み可能でなければなりません。UTOFU_REG_MEM_FLAG_READ_ONLYフラグが指定されている場合は、そのメモリ領域は少なくとも読込み可能でなければなりません。

同じメモリ領域が2つ以上のVCQを使用してワンサイド通信に使用される場合、そのメモリ領域はVCQごとに登録する必要があります。その場合、返却されるSTADD値は同じとは限りません。

ほとんどのワンサイド通信開始関数とワンサイド通信準備関数はlcl_stadd仮引数とrmt_stadd仮引数を持ちます。lcl_stadd仮引数はローカル計算ノード(origin)で割り当てられたSTADDです。rmt_stadd仮引数はリモート計算ノード(target)で割り当てられたSTADDです。そのため、通信を開始する前に、リモート計算ノードで割り当てられたSTADDはリモートプロセスによってローカルプロセスに通知されていなければなりません。STADDがあるVCQで割り当てられたら、そのSTADDはそのVCQにおける複数のワンサイド通信に使用できます。複数のワンサイド通信に同時に使用することもできます。

登録されたメモリ領域は、free()関数などによって解放する前に登録を解除されなければなりません。Put通信では、対応するTCQディスクリプタまたはローカル通知MRQディスクリプタが書かれた後に、ローカルメモリ領域の登録を解除することができます。Get通信では、対応するローカル通知MRQディスクリプタが書かれた後に、ローカルメモリ領域の登録を解除できます。どの通信においても、リモート計算ノードでリモート通知MRQディスクリプタが書かれた後か、ローカル計算ノードでローカル通知MRQディスクリプタが書かれた後に、リモートメモリ領域の登録を解除できます。

通常、STADDは、utofu_reg_mem()関数が呼ばれたときに、デバイスドライバ(Tofuドライバ)によって割り当てられます。その値はuTofu使用者には予測不可能です。しかし、utofu_reg_mem_with_stag()関数を使用して予測可能なSTADDを割り当てることもできます。STagは64ビットのSTADDの一部分のビットです。STagの特定の範囲はuTofu使用者のために予約されています。ローカルプロセスが予測可能なSTagを使用してリモートプロセスがメモリ領域を登録すれば、ローカルプロセスはutofu_query_stadd()関数を使用してそのSTADDを知ることができます。utofu_reg_mem_with_stag()関数を使用して登録したメモリ領域も、utofu_dereg_mem()関数を使用して登録を解除する必要があります。同じSTagを使用して2回以上utofu_reg_mem_with_stag()関数を呼ぶことはできません。ただし、utofu_dereg_mem()関数によって登録を解除すれば、前回と同じメモリ領域または異なるメモリ領域に対して、同じSTagをutofu_reg_mem_with_stag()関数で使用できます。

STADDは64ビット符号なし整数です。STADDに対してオフセットベースの計算をすることによって、登録したメモリ領域の一部分だけをワンサイド通信に使用できます。例えば、大きなメモリ領域の中の様々な領域を転送したい場合は、まず、領域全体を登録し、STADDとしてhead_staddを得ます。次に“head_stadd + random_offset”の値をlcl_stadd仮引数としてutofu_put()関数に指定できます。rmt_staddも同様に計算できます。

3.6.1 STADD管理関数

3.6.1.1 utofu_reg_mem

VCQにメモリ領域を登録します。

書式

```
int utofu_reg_mem(
    utofu_vcq_hdl_t    vcq_hdl,
    void              *addr,
    size_t             size,
    unsigned long int  flags,
    utofu_stadd_t     *stadd)
```

説明

登録したメモリ領域はutofu_dereg_mem()関数で解除する必要があります。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vcq_hdl	VCQハンドル。	IN
addr	メモリ領域の先頭のポインタ。	IN
size	メモリ領域のバイトサイズ。 この値は1以上である必要があります。	IN
flags	UTOFU_REG_MEM_FLAG_*のビットごとのOR。	IN
stadd	メモリ領域の先頭のSTADD。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_FULL	このVCQでこれ以上STADDを割り当てることができません。 utofu_dereg_mem()関数を呼んでから再度この関数を呼べば成功する可能性があります。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のSTADDを割り当てることができません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のSTADDはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.6.1.2 utofu_reg_mem_with_stag

STagを指定してVCQにメモリ領域を登録します。

書式

```
int utofu_reg_mem_with_stag(
    utofu_vcq_hdl_t    vcq_hdl,
    void              *addr,
    size_t             size,
    unsigned int       stag,
    unsigned long int  flags,
    utofu_stadd_t     *stadd)
```

説明

割り当てることができるSTagは0から“utofu_onesided_caps::num_reserved_stags -1”です。

登録したメモリ領域はutofu_dereg_mem()関数で解除する必要があります。

同じSTagはutofu_dereg_mem()関数によって解放されるまで再び使用することはできません。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vcq_hdl	VCQハンドル。	IN
addr	メモリ領域の先頭のポインタ。 このアドレスはutofu_onesided_caps::stag_address_alignmentの倍数である必要があります。	IN
size	メモリ領域のバイトサイズ。 サイズはutofu_onesided_caps::stag_address_alignmentの倍数である必要があります。	IN

仮引数名	説明	IN/OUT
stag	STag。 この値はutofu_onesided_caps::num_reserved_stags未満である必要があります。	IN
flags	UTOFU_REG_MEM_FLAG_*のビットごとのOR。	IN
stadd	メモリ領域の先頭のSTADD。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_USED	指定されたSTagはこのVCQですでに使用されています。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のSTagを割り当てることができません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のSTagはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.6.1.3 utofu_query_stadd

STagが割り当てられたメモリ領域のSTADDを問い合わせます。

書式

```
int utofu_query_stadd(
    utofu_vcq_id_t  vcq_id,
    unsigned int    stag,
    utofu_stadd_t  *stadd)
```

説明

ローカルプロセスとリモートプロセスのどちらのメモリ領域のSTADDでも問い合わせることができます。この関数が返却するSTADDはutofu_reg_mem_with_stag()関数が返却したSTADDと同じ値になります。

この関数は1つのSTagに対して複数回呼び出すことができます。

引数

仮引数名	説明	IN/OUT
vcq_id	utofu_reg_mem_with_stag()関数の呼出し時に使用したVCQハンドルに対応するVCQ ID。 STADDがリモートプロセスのものであれば、VCQ IDもそのリモートプロセスのものである必要があります。	IN
stag	utofu_reg_mem_with_stag()関数の呼出し時に使用したSTag。	IN
stadd	メモリ領域の先頭のSTADD。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.6.1.4 utofu_dereg_mem

VCQへのメモリ領域の登録を解除します。

書式

```
int utofu_dereg_mem(
    utofu_vcq_hdl_t   vcq_hdl,
```

```
    utofu_stadd_t     stadd,
    unsigned long int flags)
```

説明

二重解放をしてはいけません。

この関数はシステムコールの呼出しを伴うことがあります。

引数

仮引数名	説明	IN/OUT
vcq_hdl	VCQハンドル。	IN
stadd	メモリ領域の先頭のSTADD。	IN
flags	UTOFU_DEREG_MEM_FLAG_*のビットごとのOR。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_NOT_AVAILABLE	flags仮引数で指定された種類のSTADDを解除できません。
UTOFU_ERR_NOT_SUPPORTED	flags仮引数で指定された種類のSTADDはサポートされていません。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.6.2 STADDのフラグ

3.6.2.1 UTOFU_REG_MEM_FLAG_*

utofu_reg_mem()関数とutofu_reg_mem_with_stag()関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、メモリ領域の登録の動作を指定できます。

マクロ

マクロ名	説明
UTOFU_REG_MEM_FLAG_READ_ONLY	読込み専用。 このフラグは、登録するメモリ領域が、ローカルプロセスからもリモートプロセスからもワンサイド通信によって更新されないことを示します。このフラグを指定してメモリ領域を登録したにもかかわらず、そのSTADDをPutやARMWのリモートSTADDまたはGetのローカルSTADDとして使用した場合は、その通信はエラーとなりTCQまたはMRQによってエラーが通知されます。登録しようとするメモリ領域が、登録しようとしているプロセスから書込み可能ではない場合は、このフラグを指定する必要があります。あるメモリ領域が書込み可能である場合は、そのメモリ領域をこのフラグありとなしの両方で登録した状態にすることもできます。

3.6.2.2 UTOFU_DEREG_MEM_FLAG_*

utofu_dereg_mem()関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、メモリ領域の登録解除の動作を指定できます。

現在のバージョンではフラグは定義されていません。

3.7 ワンサイド通信の実行

TOQ(transmit order queue)、TCQ(transmit complete queue)、およびMRQ(message receive queue)はVCQの構成要素です。これらはTNIが書込み・読込みを行うキューです。

TOQディスクリプタはutofu_put()などの関数を通じてTOQに投入されます。TOQディスクリプタが投入されると、そのVCQに対応するTNIはそのディスクリプタを読み込み、ワンサイド通信を開始します。TOQディスクリプタにはPut、Put Piggyback、Get、およびARMW(Atomic Read Modify Write)の種類があります。

ワンサイド通信の完了は3つの方法で確認できます。

- UTOFU_ONESIDED_FLAG_TCQ_NOTICEフラグがワンサイド通信開始関数に指定された場合は、ローカル計算ノードからのパケットの送出が完了したタイミングで、ローカルVCQに対応するTCQにTCQディスクリプタが書き込まれます。PutのTOQディスクリプタに対しては、このTCQディスクリプタはローカルメモリを別のデータに書き換えても良いことを示しています。
- UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICEフラグがワンサイド通信開始関数に指定された場合は、ローカル計算ノードとリモート計算ノードの両方で通信が完了したタイミングで、ローカルVCQに対応するMRQにMRQディスクリプタが書き込まれます。PutとARMWのTOQディスクリプタに対しては、このMRQディスクリプタはリモートメモリが更新されたことを示しています。GetのTOQディスクリプタに対しては、このMRQディスクリプタは、ローカルメモリが更新され、その更新されたデータを読み込んでも良いことを示しています。ARMWのTOQディスクリプタに対しては、このMRQディスクリプタにリモートメモリの更新前の値が書かれています。
- UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICEフラグがワンサイド通信開始関数に指定された場合は、リモート計算ノードで通信が完了したタイミングで、リモートVCQに対応するMRQにMRQディスクリプタが書き込まれます。PutとARMWのTOQディスクリプタに対しては、このMRQディスクリプタはリモートメモリが更新されたことを示しています。GetのTOQディスクリプタに対しては、このMRQディスクリプタはリモートメモリを別のデータに書き換えても良いことを示しています。

パケット送出や通信が完了する前に転送元のメモリを書き換えた場合は、転送先のメモリに書き込まれる内容が、書換え前の値になるとも書換え後の値になるとも保証されません。通信が完了する前に転送先のメモリを読み込んだ場合は、読み込んだ内容が、通信前の値になるとも通信後の値になるとも保証されません。

TCQはutofu_poll_tcq()関数でポーリングできます。utofu_{put|get}_stride()関数およびutofu_{put|get}_stride_gap()関数以外のワンサイド通信開始関数では、UTOFU_ONESIDED_FLAG_TCQ_NOTICEフラグが指定されていれば、1回のワンサイド通信開始関数呼出しが1つのTCQディスクリプタに対応し、それに対してutofu_poll_tcq()関数はUTOFU_SUCCESSを返却します。utofu_{put|get}_stride()関数とutofu_{put|get}_stride_gap()関数では、UTOFU_ONESIDED_FLAG_TCQ_NOTICEフラグが指定されていれば、1回のワンサイド通信開始関数呼出しがnum_blocks仮引数で指定された値と同じ数のTCQディスクリプタに対応します。しかし、UTOFU_ONESIDED_FLAG_TCQ_NOTICEフラグの指定にかかわらず、TNIがローカル計算ノードからパケットを送出するときにエラーが発生した場合は、utofu_poll_tcq()関数はUTOFU_ERR_TCQ_*を返却します。すべてのワンサイド通信開始関数とutofu_poll_tcq()関数はcbdata仮引数を持ちます。ワンサイド通信開始関数で指定したcbdataの値がutofu_poll_tcq()関数のcbdataに設定されます。そのため、uTofu使用者は、utofu_poll_tcq()関数がUTOFU_ERR_NOT_FOUND以外の値を返却したときに、どの通信が完了したのかを特定できます。

MRQはutofu_poll_mrq()関数でポーリングできます。1回のワンサイド通信開始関数呼出しで通知されるMRQディスクリプタの数はutofu_poll_tcq()関数と同様に決まります。utofu_poll_mrq()関数はエラーなく完了した通信に対してUTOFU_SUCCESSを返却し、エラーの発生した通信に対してUTOFU_ERR_MRQ_*を返却します。完了した通信はutofu_poll_mrq()関数のnotice仮引数を使用して特定できます。

UTOFU_ONESIDED_FLAG_*_NOTICEフラグを指定しなかった場合でも、エラーが発生していないことを確認するため、また、キューに空きを作るため、utofu_poll_tcq()関数やutofu_poll_mrq()関数を呼び出す必要があります。utofu_poll_tcq()関数が十分呼ばれないためにTCQに空きがなくなった場合は、ワンサイド通信開始関数やワンサイド通信一括開始関数はUTOFU_ERR_BUSYを返却します。utofu_poll_mrq()関数が十分呼ばれないためにMRQに空きがなくなった場合は、MRQオーバーフローのエラーが発生します。TCQに十分に空きを作るには、復帰値がUTOFU_ERR_TCQ_*またはUTOFU_ERR_NOT_FOUNDとなるまでutofu_poll_tcq()関数を繰り返し呼んでください。同様に、MRQに十分に空きを作るには、復帰値がUTOFU_ERR_MRQ_*またはUTOFU_ERR_NOT_FOUNDとなるまでutofu_poll_mrq()関数を繰り返し呼んでください。

すべてのワンサイド通信開始関数はrmt_vcq_id仮引数とrmt_stadd仮引数を持ちます。utofu_reg_mem_with_stag()関数を使用して取得した値をrmt_stadd仮引数に指定する場合を除いて、これらの値は何らかの方法(例えばuTofu以外の帯域外の通信)でリモート計算ノードから取得する必要があります。

ワンサイド通信開始関数では、リモートVCQ ID(rmt_vcq_id仮引数)またはビットフラグ(flags仮引数)によって、通信経路を指定できます。もし両方で通信経路を指定した場合は、ビットフラグに指定した通信経路が優先されます。

MTUと送信間隔はutofu_*_gap()関数によって明に指定できます。ほかの関数では、MTUにはutofu_onesided_caps::max_mtuが使われ、送信間隔には0が使われます。utofu_*_gap()関数のgap仮引数には0以上utofu_onesided_caps::max_gap以下の値を指定で

きます。0を指定した場合はパケット間に送信間隔を空けません。0以外を指定した場合は、gapの値が0の場合の送出帯域を100%とすると、“8/(gap+8)”まで送出帯域を抑制します。すなわちgapの値が8の時に50%、24の時に25%の送出帯域となります。

あるメモリ領域内に同じ間隔で存在する同じサイズの複数のデータをPutまたはGetによって転送したい場合には、utofu_{put|get}_stride()関数やutofu_{put|get}_stride_gap()関数が使えます。これらの関数では、1回の関数呼出しで、PutやGetのTOQディスクリプタを複数まとめてTOQに投入します。隣り合うデータの先頭同士の間隔をストライドと呼びます。ローカルSTADDをlcl_stadd、リモートSTADDをrmt_stadd、データの長さをlength、ストライドの長さをstrideとすると、Putの場合では、1つ目のPutではlcl_staddから“lcl_stadd + length”の範囲をrmt_staddから“rmt_stadd + length”の範囲に、2つ目のPutでは“lcl_stadd + stride”から“lcl_stadd + stride + length”の範囲を“rmt_stadd + stride”から“rmt_stadd + stride + length”の範囲に、3つ目のPutでは“lcl_stadd + (stride * 2)”から“lcl_stadd + (stride * 2) + length”の範囲を“rmt_stadd + (stride * 2)”から“rmt_stadd + (stride * 2) + length”の範囲に、というように転送します。

すべてのワンサイド通信開始関数utofu_*()には、対応するワンサイド通信準備関数utofu_prepare_*()が存在します。これらの関数はTOQディスクリプタをTOQに投入せず、代わりに関数呼出し元が指定したメモリにTOQディスクリプタを書き込みます。書き込まれたディスクリプタはワンサイド通信一括開始関数utofu_post_toq()によってTOQに投入できます。これらの関数は、同じTOQディスクリプタを繰返し投入するときに、TOQディスクリプタを作成するコストを削減するために使用できます。

3.7.1 ワンサイド通信開始関数

3.7.1.1 utofu_put

PutディスクリプタをTOQに投入します。

書式

```
int utofu_put(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ読込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.2 utofu_put_gap

送信間隔付きのPutディスクリプタをTOQに投入します。

書式

```
int utofu_put_gap(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    uint64_t            edata,
    unsigned long int   flags,
    size_t              mtu,
    size_t              gap,
    void               *cbdata)
```

説明

mtuとgap以外の仮引数はutofu_put()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ読込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
mtu	パケットのMTU(最大転送サイズ)。 utofu_onesided_caps::max_mtu以下の値を指定できます。	IN
gap	パケットの間に挿入する送信間隔。 utofu_onesided_caps::max_gap以下の値を指定できます。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかの UTOFU_ERR_*エラー。

3.7.1.3 utofu_put_stride

複数のPutディスクリプタをTOQに投入します。

書式

```
int utofu_put_stride(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    size_t              stride,
    size_t              num_blocks,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

説明

strideとnum_blocks以外の仮引数はutofu_put()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ読込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
stride	バイト単位でのストライドの長さ。	IN
num_blocks	Putディスクリプタの数。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。

値	説明
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.4 utofu_put_stride_gap

複数の送信間隔付きのPutディスクリプタをTOQに投入します。

書式

```
int utofu_put_stride_gap(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    size_t              stride,
    size_t              num_blocks,
    uint64_t            edata,
    unsigned long int   flags,
    size_t              mtu,
    size_t              gap,
    void               *cbdata)
```

説明

stride、num_blocks、mtu、gap以外の仮引数はutofu_put()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ読込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスで utofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
stride	バイト単位でのストライドの長さ。	IN
num_blocks	Putディスクリプタの数。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
mtu	パケットのMTU(最大転送サイズ)。 utofu_onesided_caps::max_mtu以下の値を指定できます。	IN
gap	パケットの間に挿入する送信間隔。 utofu_onesided_caps::max_gap以下の値を指定できます。	IN

仮引数名	説明	IN/OUT
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.5 utofu_put_piggyback

Put Piggybackディスクリプタを TOQに投入します。

書式

```
int utofu_put_piggyback(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    void               *lcl_data,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

説明

utofu_onesided_caps::max_piggyback_sizeバイト以下のデータを転送できます。

lcl_data以外の仮引数はutofu_put()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_data	ローカルデータへのポインタ。 この関数が復帰したらこのメモリを別のデータに安全に書き換えられます。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_piggyback_size以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.6 utofu_put_piggyback8

8バイト以下用のPut PiggybackディスクリプタをTOQに投入します。

書式

```
int utofu_put_piggyback8(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    uint64_t            lcl_data,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

説明

8バイト以下のデータを転送できます。データが8バイト未満の場合はlcl_dataの下位ビット側のデータが転送されます。

lcl_data以外の仮引数はutofu_put()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_data	ローカルデータの値。	IN
rmt_stadd	リモート(データ書込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 8以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.7 utofu_get

Getディスクリプタを TOQに投入します。

書式

```
int utofu_get(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ書込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ読込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.8 utofu_get_gap

送信間隔付きのGetディスクリプタをTOQに投入します。

書式

```
int utofu_get_gap(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
```

```
    uint64_t            edata,
    unsigned long int   flags,
    size_t              mtu,
    size_t              gap,
    void               *cbdata)
```

説明

mtuとgap以外の仮引数はutofu_get()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ書込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ読込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
mtu	パケットのMTU(最大転送サイズ)。 utofu_onesided_caps::max_mtu以下の値を指定できます。	IN
gap	パケットの間に挿入する送信間隔。 utofu_onesided_caps::max_gap以下の値を指定できます。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.9 utofu_get_stride

複数のGetディスクリプタをTOQに投入します。

書式

```
int utofu_get_stride(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
```

```
    size_t              stride,
    size_t              num_blocks,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

説明

strideとnum_blocks以外の仮引数はutofu_get()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ書込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ読込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
stride	バイト単位でのストライドの長さ。	IN
num_blocks	Getディスクリプタの数。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.10 utofu_get_stride_gap

複数の送信間隔付きのGetディスクリプタをTOQに投入します。

書式

```
int utofu_get_stride_gap(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    utofu_stadd_t       lcl_stadd,
    utofu_stadd_t       rmt_stadd,
    size_t              length,
    size_t              stride,
    size_t              num_blocks,
    uint64_t            edata,
```

```
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *cbdata)
```

説明

stride、num_blocks、mtu、gap以外の仮引数はutofu_get()関数と同じです。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
lcl_stadd	ローカル(データ書込み)メモリ領域の開始アドレスのSTADD。 このプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
rmt_stadd	リモート(データ読込み)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。	IN
length	バイト単位でのデータの長さ。 utofu_onesided_caps::max_putget_size以下の値を指定できます。	IN
stride	バイト単位でのストライドの長さ。	IN
num_blocks	Getディスクリプタの数。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
mtu	パケットのMTU(最大転送サイズ)。 utofu_onesided_caps::max_mtu以下の値を指定できます。	IN
gap	パケットの間に挿入する送信間隔。 utofu_onesided_caps::max_gap以下の値を指定できます。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.11 utofu_armw4

コンペア・アンド・スワップを除く32ビットARMWディスクリプタをTOQに投入します。

書式

```
int utofu_armw4(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
```

```
    uint32_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
armw_op	ARMWの演算種別。	IN
op_value	ローカルオペランド。	IN
rmt_stadd	リモート(データ更新)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。リモートでのメモリアドレスは4バイトに整列されている必要があります。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.12 utofu_armw8

コンペア・アンド・スワップを除く64ビットARMWディスクリプタをTOQに投入します。

書式

```
int utofu_armw8(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint64_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。	IN

仮引数名	説明	IN/OUT
	リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	
armw_op	ARMWの演算種別。	IN
op_value	ローカルオペランド。	IN
rmt_stadd	リモート(データ更新)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。リモートでのメモリアドレスは8バイトに整列されている必要があります。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.13 utofu_cswap4

コンペア・アンド・スワップの32ビットARMWディスクリプタをTOQに投入します。

書式

```
int utofu_cswap4(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    uint32_t            old_value,
    uint32_t            new_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
old_value	古い値(比較オペランド)。	IN
new_value	新しい値(書込みオペランド)。	IN
rmt_stadd	リモート(データ更新)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。リモートでのメモリアドレスは4バイトに整列されている必要があります。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN

仮引数名	説明	IN/OUT
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.14 utofu_cswap8

コンペア・アンド・スワップの64ビットARMWディスクリプタをTOQに投入します。

書式

```
int utofu_cswap8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           old_value,
    uint64_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
rmt_vcq_id	リモートVCQ ID。 リモートプロセスでutofu_query_vcq_id()関数を呼ぶことによって得られます。	IN
old_value	古い値(比較オペランド)。	IN
new_value	新しい値(書込みオペランド)。	IN
rmt_stadd	リモート(データ更新)メモリ領域の開始アドレスのSTADD。 リモートプロセスでutofu_reg_mem()関数を呼ぶことによって得られます。リモートでのメモリアドレスは8バイトに整列されている必要があります。	IN
edata	utofu_poll_mrq()関数によって渡されるEDATA。 utofu_onesided_caps::max_edata_sizeバイト以下で値を指定できます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。

値	説明
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.1.15 utofu_nop

NOPディスクリプタをTOQに投入します。

書式

```
int utofu_nop(
    utofu_vcq_hdl_t    vcq_hdl,
    unsigned long int  flags,
    void               *cbdata)
```

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
flags	UTOFU_ONESIDED_FLAG_*のビットごとのOR。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.2 ワンサイド通信準備関数

ワンサイド通信準備関数は、関数呼出し元が指定したメモリにTOQディスクリプタを書き込みます。

すべてのワンサイド通信準備関数utofu_prepare_*()関数には対応するワンサイド通信開始関数utofu_*()が存在します。仮引数は、対応するワンサイド通信開始関数に対して、cbdataが削除され、descとdesc_sizeが追加されています。

desc仮引数とdesc_size仮引数および復帰値は共通で以下のとおりであり、これ以降の個別のワンサイド通信準備関数の説明では引数と復帰値の説明を省略します。

引数

仮引数名	説明	IN/OUT
desc	作成されたディスクリプタ。 メモリのバイトサイズは、utofu_prepare_*_stride()関数では“utofu_onesided_caps::max_toq_desc_size * num_blocks”以上、utofu_prepare_*_stride以外の関数ではutofu_onesided_caps::max_toq_desc_size以上である必要があります。メモリアドレスは8バイトに整列されている必要があります。	OUT
desc_size	作成されたディスクリプタのバイトサイズ。 値は“utofu_onesided_caps::max_toq_desc_size * num_blocks”やutofu_onesided_caps::max_toq_desc_sizeより小さい場合があります。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。

値	説明
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.2.1 utofu_prepare_put

Putディスクリプタを準備します。

書式

```
int utofu_prepare_put(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.2 utofu_prepare_put_gap

送信間隔付きのPutディスクリプタを準備します。

書式

```
int utofu_prepare_put_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.3 utofu_prepare_put_stride

複数のPutディスクリプタを準備します。

書式

```
int utofu_prepare_put_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.4 utofu_prepare_put_stride_gap

複数の送信間隔付きのPutディスクリプタを準備します。

書式

```
int utofu_prepare_put_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.5 utofu_prepare_put_piggyback

Put Piggybackディスクリプタを準備します。

書式

```
int utofu_prepare_put_piggyback(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    void              *lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.6 utofu_prepare_put_piggyback8

8バイト以下用のPut Piggybackディスクリプタを準備します。

書式

```
int utofu_prepare_put_piggyback8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           lcl_data,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.7 utofu_prepare_get

Getディスクリプタを準備します。

書式

```
int utofu_prepare_get(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
```

```
    unsigned long int  flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.8 utofu_prepare_get_gap

送信間隔付きのGetディスクリプタを準備します。

書式

```
int utofu_prepare_get_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.9 utofu_prepare_get_stride

複数のGetディスクリプタを準備します。

書式

```
int utofu_prepare_get_stride(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.10 utofu_prepare_get_stride_gap

複数の送信間隔付きのGetディスクリプタを準備します。

書式

```
int utofu_prepare_get_stride_gap(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    utofu_stadd_t      lcl_stadd,
    utofu_stadd_t      rmt_stadd,
    size_t             length,
    size_t             stride,
    size_t             num_blocks,
    uint64_t           edata,
    unsigned long int  flags,
    size_t             mtu,
    size_t             gap,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.11 utofu_prepare_armw4

コンペア・アンド・スワップを除く32ビットARMWディスクリプタを準備します。

書式

```
int utofu_prepare_armw4(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint32_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.12 utofu_prepare_armw8

コンペア・アンド・スワップを除く64ビットARMWディスクリプタを準備します。

書式

```
int utofu_prepare_armw8(
    utofu_vcq_hdl_t     vcq_hdl,
    utofu_vcq_id_t      rmt_vcq_id,
    enum utofu_armw_op  armw_op,
    uint64_t            op_value,
    utofu_stadd_t       rmt_stadd,
    uint64_t            edata,
    unsigned long int   flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.13 utofu_prepare_cswap4

コンペア・アンド・スワップの32ビットARMWディスクリプタを準備します。

書式

```
int utofu_prepare_cswap4(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint32_t           old_value,
    uint32_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
    unsigned long int  flags,
    void              *desc,
    size_t            *desc_size)
```

3.7.2.14 utofu_prepare_cswap8

コンペア・アンド・スワップの64ビットARMWディスクリプタを準備します。

書式

```
int utofu_prepare_cswap8(
    utofu_vcq_hdl_t    vcq_hdl,
    utofu_vcq_id_t     rmt_vcq_id,
    uint64_t           old_value,
    uint64_t           new_value,
    utofu_stadd_t      rmt_stadd,
    uint64_t           edata,
```

```
    unsigned long int  flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.2.15 utofu_prepare_nop

NOPディスクリプタを準備します。

書式

```
int utofu_prepare_nop(
    utofu_vcq_hdl_t    vcq_hdl,
    unsigned long int  flags,
    void               *desc,
    size_t             *desc_size)
```

3.7.3 ワンサイド通信一括開始関数

3.7.3.1 utofu_post_toq

ディスクリプタをTOQに投入して通信を一括して開始します。

書式

```
int utofu_post_toq(
    utofu_vcq_hdl_t  vcq_hdl,
    void            *desc,
    size_t           desc_size,
    void            *cbdata)
```

説明

この関数はディスクリプタをTOQに投入してそのワンサイド通信を開始します。投入するTOQディスクリプタはutofu_prepare_*()関数によって作成できます。

utofu_prepare_*()関数を複数回呼ぶことによって連続したメモリ領域に複数のTOQディスクリプタを書き込んでおき、この関数にそのメモリ領域を渡すことで、複数の通信を一括して開始することができます。異なる種類の通信を混在させることもできます。

この関数はUTOFU_ONESIDED_FLAG_DELAY_STARTフラグによってまだ開始されていなかったワンサイド通信も開始します。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。 このプロセスでutofu_create_vcq()関数を呼ぶことによって得られます。	IN
desc	utofu_prepare_*()関数によって作成されたディスクリプタ。 utofu_prepare_*()関数に指定されたすべてのローカルVCQハンドルは、この関数に指定されたVCQハンドルと同じである必要があります。desc_sizeが0の場合は、この引数は無視されます。	IN
desc_size	ディスクリプタの合計バイトサイズ。 0を指定することで、UTOFU_ONESIDED_FLAG_DELAY_STARTフラグのためにまだ開始されていないワンサイド通信だけを開始できます。	IN
cbdata	utofu_poll_tcq()関数によって渡されるコールバックデータ。 desc_sizeが0の場合は、この引数は無視されます。複数のTOQディスクリプタを投入した場合は、すべての通信に同じコールバックデータが適用されます。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。

値	説明
UTOFU_ERR_BUSY	TOQに空きがありません。 utofu_poll_tcq()関数を呼んでから再度この関数を呼ぶ必要があります。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.4 ワンサイド通信完了確認関数

3.7.4.1 utofu_poll_tcq

TCQをポーリングします。

書式

```
int utofu_poll_tcq(
    utofu_vcq_hdl_t     vcq_hdl,
    unsigned long int   flags,
    void              **cbdata)
```

説明

指定されたVCQに新しいTCQディスクリプタが存在したら、この関数はUTOFU_SUCCESSを返却してcbdataを設定します。指定されたVCQに新しいTCQディスクリプタが存在しなかったら、この関数はUTOFU_ERR_NOT_FOUNDを返却します。

TCQディスクリプタの順番はUTOFU_ONESIDED_FLAG_TCQ_NOTICEフラグが指定されなかったTOQディスクリプタと同じ順番です。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。	IN
flags	UTOFU_POLL_FLAG_*のビットごとのOR。	IN
cbdata	utofu_put()関数などに渡されたコールバックデータを書き込むメモリ領域へのポインタ。 コールバックデータはこの関数がUTOFU_ERR_NOT_FOUND以外の値を返却するときだけ設定されます。	OUT

復帰値

値	説明
UTOFU_SUCCESS	エラーなしで完了した通信のTCQディスクリプタが見つかりました。
UTOFU_ERR_NOT_FOUND	新しいTCQディスクリプタは見つかりませんでした。
UTOFU_ERR_TCQ_*	エラーを伴う通信のTCQディスクリプタが見つかりました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.4.2 utofu_poll_mrq

MRQをポーリングします。

書式

```
int utofu_poll_mrq(
    utofu_vcq_hdl_t          vcq_hdl,
    unsigned long int        flags,
    struct utofu_mrq_notice *notice)
```

説明

指定されたVCQに新しいMRQディスクリプタが存在したら、この関数はUTOFU_SUCCESSを返却してnoticeにデータを書き込みます。指定されたVCQに新しいMRQディスクリプタが存在しなかったら、この関数はUTOFU_ERR_NOT_FOUNDを返却します。

この関数ではローカル通知MRQディスクリプタとリモート通知MRQディスクリプタの両方を扱います。完了した通信の種別はnoticeによって確認できます。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。	IN
flags	UTOFU_POLL_FLAG_*のビットごとのOR。	IN
notice	MRQ通知。 MRQ通知はこの関数がUTOFU_ERR_NOT_FOUND以外の値を返却するときだけ設定されます。	OUT

復帰値

値	説明
UTOFU_SUCCESS	エラーなしで完了した通信のMRQディスクリプタが見つかりました。
UTOFU_ERR_NOT_FOUND	新しいMRQディスクリプタは見つかりませんでした。
UTOFU_ERR_MRQ_*	エラーを伴う通信のMRQディスクリプタが見つかりました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.5 ワンサイド通信問合せ関数

3.7.5.1 utofu_query_num_unread_tcq

未読のTCQディスクリプタの数を問い合わせます。

書式

```
int utofu_query_num_unread_tcq(
    utofu_vcq_hdl_t  vcq_hdl,
    size_t          *count)
```

説明

この関数は現在のTNIの稼働状況を見積もるために使用できます。

この関数は、対応するTCQディスクリプタをuTofu実装がまだ読んでいないTOQディスクリプタの数を返却します。この数は、実行が予定されているがまだ完了していないワンサイド通信の数に、おおよそ一致します。この関数が0を返却した場合は、次に呼び出すワンサイド通信開始関数は通信をすぐに開始できます。

この関数はVCQハンドルを仮引数に持ちますが、ディスクリプタ数はCQごとに数えられます。これはTNIがCQ単位でTOQディスクリプタを処理するためです。したがって、CQが複数のVCQで共有されている場合は、返却された値はVCQ単位で管理している数と異なることがあります。

引数

仮引数名	説明	IN/OUT
vcq_hdl	ローカルVCQハンドル。	IN
count	未読のTCQディスクリプタの数。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.7.6 通信完了通知を示す構造体

3.7.6.1 struct utofu_mrq_notice

MRQ通知の内容。

定義

```
struct utofu_mrq_notice {
    uint8_t        notice_type;
    uint8_t        padding1[7];
    utofu_vcq_id_t vcq_id;
    uint64_t       edata;
    uint64_t       rmt_value;
    utofu_stadd_t  lcl_stadd;
    utofu_stadd_t  rmt_stadd;
    uint64_t       reserved[2];
}
```

説明

utofu_poll_mrq()関数がこの構造体を設定します。

メンバ

メンバ名	説明
notice_type	MRQ通知の種別。 値はutofu_mrq_notice_type列挙型の列挙定数UTOFU_MRQ_TYPE_*のどれかです。
padding1	サイズ調整用フィールド。 このフィールドを使用してはいけません。
vcq_id	通信相手のVCQ ID。 この値は、ローカル通知ではリモート(target) VCQ ID、リモート通知ではローカル(origin) VCQ IDになります。埋め込まれている通信経路座標は、utofu_query_vcq_id()関数やutofu_construct_vcq_id()関数で得られるものと同じ、計算ノードのA、B、C座標です。そのため、utofu_set_vcq_id_path()関数によって別のデフォルト通信経路座標をVCQ IDに埋め込んだ場合は、ワンサイド通信の実行時に指定したリモートVCQ IDとは異なる値になることがあります。
edata	対応するワンサイド通信開始関数やワンサイド通信準備関数で指定したEDATAの値。
rmt_value	更新前のリモートメモリの値。 32ビットARMWの場合は上位32ビットが0に設定されます。このフィールドはUTOFU_MRQ_TYPE_LCL_ARMWのMRQ通知だけで設定されます。
lcl_stadd	ローカル(origin) STADD。 このSTADDは、Put/Getではデータの末尾の1バイト先(TOQディスクリプタ投入時の“lcl_stadd + length”)を指し、ARMWではデータの先頭(TOQディスクリプタ投入時のlcl_stadd)を指します。このフィールドはUTOFU_MRQ_TYPE_{LCL|RMT}_GETのMRQ通知だけで設定されます。
rmt_stadd	リモート(target) STADD。 このSTADDは、Put/Getではデータの末尾の1バイト先(TOQディスクリプタ投入時の“rmt_stadd + length”)を指し、ARMWではデータの先頭(TOQディスクリプタ投入時のrmt_stadd)を指します。このフィールドはUTOFU_MRQ_TYPE_{LCL|RMT}_PUT、UTOFU_MRQ_TYPE_RMT_GET、およびUTOFU_MRQ_TYPE_{LCL|RMT}_ARMWのMRQ通知だけで設定されます。
reserved	将来の拡張のための予約フィールド。 このフィールドを使用してはいけません。

3.7.7 ARMW演算の種別

3.7.7.1 enum utofu_armw_op

ワンサイド通信のARMW演算の種別。

説明

これらの列挙定数は、ワンサイド通信を開始するutofu_armw4()、utofu_armw8()、およびそれらに対応するutofu_prepare_*()関数の実引数に使用されます。

列挙定数

列挙定数	説明
UTOFU_ARMW_OP_SWAP	スワップ。
UTOFU_ARMW_OP_ADD	符号なし整数として加算。
UTOFU_ARMW_OP_XOR	ビットごとの排他的論理和。
UTOFU_ARMW_OP_AND	ビットごとの論理積。
UTOFU_ARMW_OP_OR	ビットごとの論理和。

3.7.8 通信完了通知の種別

3.7.8.1 enum utofu_mrq_notice_type

MRQディスクリプタの種別。

説明

これらの列挙定数は、utofu_poll_mrq()関数の呼出しによって新しいMRQディスクリプタが見つかったときに、utofu_mrq_notice::notice_typeに設定されます。

列挙定数

列挙定数	説明
UTOFU_MRQ_TYPE_LCL_PUT	Put通信のローカルMRQ通知。
UTOFU_MRQ_TYPE_RMT_PUT	Put通信のリモートMRQ通知。
UTOFU_MRQ_TYPE_LCL_GET	Get通信のローカルMRQ通知。
UTOFU_MRQ_TYPE_RMT_GET	Get通信のリモートMRQ通知。
UTOFU_MRQ_TYPE_LCL_ARMW	ARMW通信のローカルMRQ通知。
UTOFU_MRQ_TYPE_RMT_ARMW	ARMW通信のリモートMRQ通知。

3.7.9 ワンサイド通信のフラグ

3.7.9.1 UTOFU_ONESIDED_FLAG_*

ワンサイド通信開始関数やワンサイド通信準備関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、ワンサイド通信の動作を指定できます。

マクロ

マクロ名	説明
UTOFU_ONESIDED_FLAG_TCQ_NOTICE	TCQ通知。 このフラグは、対応するTCQ通知をutofu_poll_tcq()関数が返却するように指示します。
UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE	リモートMRQ通知。 このフラグは、対応するリモートMRQ通知をutofu_poll_mrq()関数が返却するように指示します。
UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE	ローカルMRQ通知。

マクロ名	説明
	このフラグは、対応するローカルMRQ通知をutofu_poll_mrq()関数が返却するように指示します。
UTOFU_ONESIDED_FLAG_STRONG_ORDER	ストロングオーダ。 このフラグは、このワンサイド通信によるメモリからの読込みまたはメモリへの書込みが、ローカルVCQ・リモートVCQ・通信経路が同じである今までのワンサイド通信によるメモリからの読込みまたはメモリへの書込みの後に行われるように指示します。
UTOFU_ONESIDED_FLAG_CACHE_INJECTION	キャッシュインジェクション。 このフラグは、ローカル・リモートTNIがデータをメモリだけでなくCPUの最終レベルキャッシュにも書き込むように指示します。
UTOFU_ONESIDED_FLAG_PADDING	キャッシュラインパディング。 このフラグは、リモートTNIがPut Piggybackの対象の領域ではないが対象のキャッシュラインに含まれる領域に不定値を書き込むように指示します。このフラグは、Put Piggybackでキャッシュインジェクションを使用するために、UTOFU_ONESIDED_FLAG_CACHE_INJECTIONフラグを指定したPut Piggybackだけに使用します。
UTOFU_ONESIDED_FLAG_DELAY_START	通信開始の先延ばし。 このフラグは、ワンサイド通信開始関数に対して、可能であれば通信をすぐには開始せずに保留状態にすることを指示します。同じVCQに対するこのフラグを指定していないワンサイド通信開始関数の次の呼出し、または同じVCQに対するutofu_post_toq()関数の次の呼出しによって、保留された通信が開始されます。保留されたワンサイド通信によってTOQに空きがないなどの理由によって、このフラグは無視されることもあります。通信の開始には一定のコストがかかるため、ワンサイド通信を繰り返す場合には、このフラグによって処理を高速化できます。このフラグが指定されていても通信が開始されることがあるため、関数を呼んだときには通信の準備ができている必要があります。例えば、リモートVCQは作成されていなければならず、転送元のデータはメモリに書き込まれていなければなりません。ワンサイド通信準備関数は通信を開始しないため、このフラグはそれらの関数に対して効果を持ちません。

3.7.9.2 UTOFU_ONESIDED_FLAG_PATH

ワンサイド通信開始関数やワンサイド通信準備関数のflags仮引数で通信経路を指定するためのビットフラグ。

マクロ

```
UTOFU_ONESIDED_FLAG_PATH(path_id)
```

説明

ワンサイド通信開始関数やワンサイド通信準備関数の呼出しでは、リモートVCQ ID(rmt_vcq_id仮引数)またはビットフラグ(flags仮引数)によって、通信経路を指定できます。

この関数形式マクロは、ワンサイド通信開始関数やワンサイド通信準備関数のflags仮引数に指定できる通信経路の値を作成します。ほかのUTOFU_ONESIDED_FLAG_*マクロとビットごとのORをとってflags仮引数に指定できます。

引数

仮引数名	説明	IN/OUT
path_id	utofu_get_path_id()関数によって作成された通信経路ID。	IN

3.7.9.3 UTOFU_ONESIDED_FLAG_SPS

ワンサイド通信開始関数やワンサイド通信準備関数のflags仮引数でセッションオフセット前進量(SPS)を指定するためのビットフラグ。

マクロ

```
UTOFU_ONESIDED_FLAG_SPS(sps)
```

説明

セッションモードVCQとして作成されたリモートVCQに対するワンサイド通信開始関数やワンサイド通信準備関数の呼出しでは、flags仮引数にセッションオフセット前進量(SPS)を指定しなければなりません。

この関数形式マクロは、ワンサイド通信開始関数やワンサイド通信準備関数のflags仮引数に指定できるSPSの値を作成します。ほかのUTOFU_ONESIDED_FLAG_*マクロとビットごとのORをとってflags仮引数に指定できます。

リモートVCQがセッションモードVCQとして作成されたものでない場合は、このフラグは無視されます。リモートVCQがセッションモードVCQとして作成されたものであるがflags仮引数にSPSが指定されなかった場合は、SPSは0として扱われます。

引数

仮引数名	説明	IN/OUT
sps	セッションモードのSPS。	IN

3.7.10 ポーリングのフラグ

3.7.10.1 UTOFU_POLL_FLAG_*

ワンサイド通信完了確認関数やバリア通信完了確認関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、ポーリング関数の動作を指定できます。

現在のバージョンではフラグは定義されていません。

3.8 バリア通信の実行

バリア通信は、バリア通信開始関数を呼ぶことによって開始され、バリア通信完了確認関数を呼ぶことによって完了したかどうかを確認できます。あるバリア通信開始関数によって開始したバリア通信は、それに対応するバリア通信完了確認関数によって完了を確認する必要があります。バリア同期と同時にリダクション演算を実行する場合は、バリア通信開始関数に入力値を指定し、バリア通信完了確認関数で出力値(演算結果)を得ます。

utofu_set_vbg()関数によって構築した1つのバリア回路では同時に1つのバリア通信を実行できます。次のバリア通信を開始する前に、前のバリア通信の完了を確認しておく必要があります。複数のバリア回路を使用すれば、同時に複数のバリア通信を実行できます。

1回のバリア通信では、そのバリア通信に参加するすべてのプロセスが、同一のバリア通信開始関数を呼び、同一のリダクション演算種別を指定する必要があります。プロセス間で関数または演算種別が異なる場合は、対応するバリア通信完了確認関数がUTOFU_ERR_BARRIER_MISMATCHを返却します。

どのバリア回路を使用するかは、バリア通信開始関数やバリア通信完了確認関数の第1引数vbg_idで指定します。この引数はutofu_alloc_vbg()関数が返却したローカルVBG IDの集合の最初のVBG IDである必要があります。このVBGは始点・終点BGに対応します。

3.8.1 バリア通信開始関数

3.8.1.1 utofu_barrier

バリア回路でバリア同期を開始します。

書式

```
int utofu_barrier(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags)
```

説明

この関数はバリア同期の完了を待たずに復帰します。完了はutofu_poll_barrier()関数で確認できます。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
flags	UTOFU_BARRIER_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がすでに開始されています。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.1.2 utofu_reduce_uint64

バリア回路でuint64_t型の値のリダクション演算を開始します。

書式

```
int utofu_reduce_uint64(
    utofu_vbg_id_t       vbg_id,
    enum utofu_reduce_op op,
    uint64_t             data[],
    size_t               num_data,
    unsigned long int    flags)
```

説明

この関数はリダクション演算の完了を待たずに復帰します。完了はutofu_poll_reduce_uint64()関数で確認できます。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
op	リダクション演算の種別。 uint64_t用の演算だけが指定可能です。バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
data	リダクション演算の入力データの配列。 配列長はnum_dataである必要があります。	IN
num_data	リダクション演算の入力データの数。 1以上utofu_barrier_caps::max_uint64_reduction以下の値を指定できます。バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
flags	UTOFU_BARRIER_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がすでに開始されています。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.1.3 utofu_reduce_double

バリア回路でdouble型の値のリダクション演算を開始します。

書式

```
int utofu_reduce_double(
    utofu_vbg_id_t       vbg_id,
    enum utofu_reduce_op op,
    double               data[],
    size_t               num_data,
    unsigned long int    flags)
```

説明

この関数はリダクション演算の完了を待たずに復帰します。完了はutofu_poll_reduce_double()関数で確認できます。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
op	リダクション演算の種別。 double用の演算だけが指定可能です。バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
data	リダクション演算の入力データの配列。 配列長はnum_dataである必要があります。	IN
num_data	リダクション演算の入力データの数。 1以上utofu_barrier_caps::max_double_reduction以下の値を指定できます。バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
flags	UTOFU_BARRIER_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がすでに開始されています。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.2 バリア通信完了確認関数

3.8.2.1 utofu_poll_barrier

バリア同期の完了をポーリングします。

書式

```
int utofu_poll_barrier(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags)
```

説明

この関数を呼び出す前に、utofu_barrier()関数によってバリア同期が開始されている必要があります。

この関数はバリア同期の完了状態にかかわらず復帰します。バリア同期が完了していた場合はこの関数はUTOFU_SUCCESSを返却します。バリア同期が完了していなかった場合はこの関数は即座にUTOFU_ERR_NOT_COMPLETEDを返却します。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
flags	UTOFU_POLL_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN

復帰値

値	説明
UTOFU_SUCCESS	完了しました。
UTOFU_ERR_NOT_COMPLETED	完了していません。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がまだ開始されていません。
UTOFU_ERR_BARRIER_*	バリア通信のエラーが発生しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.2.2 utofu_poll_reduce_uint64

uint64_t型の値のリダクション演算の完了をポーリングします。

書式

```
int utofu_poll_reduce_uint64(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags,
    uint64_t          data[])
```

説明

この関数を呼び出す前に、utofu_reduce_uint64()関数によってリダクション演算が開始されている必要があります。

この関数はリダクション演算の完了状態にかかわらず復帰します。リダクション演算が完了していた場合はこの関数はUTOFU_SUCCESSを返却します。リダクション演算が完了していなかった場合はこの関数は即座にUTOFU_ERR_NOT_COMPLETEDを返却します。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
flags	UTOFU_POLL_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
data	uint64_tのリダクション結果データ。	OUT

仮引数名	説明	IN/OUT
	配列長はutofu_reduce_uint64()関数に指定したnum_data以上である必要があり、最初のnum_data要素だけが更新されます。配列の内容はこの関数がUTOFU_SUCCESSを返却するときだけ更新されます。	

復帰値

値	説明
UTOFU_SUCCESS	完了しました。
UTOFU_ERR_NOT_COMPLETED	完了していません。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がまだ開始されていません。
UTOFU_ERR_BARRIER_*	バリア通信のエラーが発生しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.2.3 utofu_poll_reduce_double

double型の値のリダクション演算の完了をポーリングします。

書式

```
int utofu_poll_reduce_double(
    utofu_vbg_id_t    vbg_id,
    unsigned long int flags,
    double            data[])
```

説明

この関数を呼び出す前に、utofu_reduce_double()関数によってリダクション演算が開始されている必要があります。

この関数はリダクション演算の完了状態にかかわらず復帰します。リダクション演算が完了していた場合はこの関数はUTOFU_SUCCESSを返却します。リダクション演算が完了していなかった場合はこの関数は即座にUTOFU_ERR_NOT_COMPLETEDを返却します。

引数

仮引数名	説明	IN/OUT
vbg_id	最初のローカルVBG ID。 このプロセスでutofu_alloc_vbg()関数を呼ぶことによって得られます。	IN
flags	UTOFU_POLL_FLAG_*のビットごとのOR。 バリア通信に参加するすべてのプロセスで同じ値を指定する必要があります。	IN
data	doubleのリダクション結果データ。 配列長はutofu_reduce_double()関数に指定したnum_data以上である必要があり、最初のnum_data要素だけが更新されます。配列の内容はこの関数がUTOFU_SUCCESSを返却するときだけ更新されます。	OUT

復帰値

値	説明
UTOFU_SUCCESS	完了しました。
UTOFU_ERR_NOT_COMPLETED	完了していません。
UTOFU_ERR_BUSY	このバリア回路でバリア通信がまだ開始されていません。
UTOFU_ERR_BARRIER_*	バリア通信のエラーが発生しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.8.3 リダクション演算の種別

3.8.3.1 enum utofu_reduce_op

バリア通信のリダクション演算の種別。

説明

これらの列挙定数は、バリア通信を開始するutofu_reduce_uint64()関数およびutofu_reduce_double()関数の実引数に使用されます。

列挙定数

列挙定数	説明
UTOFU_REDUCE_OP_BARRIER	バリア同期(リダクション演算なし)。
UTOFU_REDUCE_OP_BAND	uint64_t型の値のビットごとの論理積。
UTOFU_REDUCE_OP_BOR	uint64_t型の値のビットごとの論理和。
UTOFU_REDUCE_OP_BXOR	uint64_t型の値のビットごとの排他的論理和。
UTOFU_REDUCE_OP_MAX	uint64_t型の値の符号なし整数としての最大値。
UTOFU_REDUCE_OP_MAXLOC	uint64_t型の値の符号なし整数としての最大値とその位置。
UTOFU_REDUCE_OP_SUM	uint64_t型の値の符号なし整数としての和。
UTOFU_REDUCE_OP_BFPSUM	double型の値の浮動小数点としての和。

3.8.4 バリア通信のフラグ

3.8.4.1 UTOFU_BARRIER_FLAG_*

バリア通信開始関数のflags仮引数に指定するためのビットフラグ。

説明

これらのフラグを使用して、バリア通信の動作を指定できます。

現在のバージョンではフラグは定義されていません。

3.9 補助機能

3.9.1 バージョン情報問合せ関数

3.9.1.1 utofu_query_tofu_version

Tofuインターコネクトのバージョンを問い合わせます。

書式

```
void utofu_query_tofu_version(
    int *major_ver,
    int *minor_ver)
```

説明

返却されるバージョンの値については、“第6章 システムの情報”を参照してください。

この関数が返すバージョンの値はuTofuでの管理上の値であり、Tofuインターコネクトそのもののバージョンとは異なります。

引数

仮引数名	説明	IN/OUT
major_ver	majorバージョン。	OUT

仮引数名	説明	IN/OUT
minor_ver	minorバージョン。	OUT

3.9.1.2 utofu_query_utofu_version

uTofuの実行時APIバージョンを問い合わせます。

書式

```
void utofu_query_utofu_version(
    int *major_ver,
    int *minor_ver)
```

引数

仮引数名	説明	IN/OUT
major_ver	majorバージョン。	OUT
minor_ver	minorバージョン。	OUT

3.9.2 計算ノード情報問合せ関数

3.9.2.1 utofu_query_my_coords

Tofuネットワーク上でのこの計算ノードの座標を問い合わせます。

書式

```
int utofu_query_my_coords(
    uint8_t coords[])
```

引数

仮引数名	説明	IN/OUT
coords	計算ノード座標(X、Y、Z、A、B、Cの順)。 配列長は6以上である必要があります。	OUT

復帰値

値	説明
UTOFU_SUCCESS	処理に成功しました。
そのほか	そのほかのUTOFU_ERR_*エラー。

3.9.3 バージョン情報のマクロ

3.9.3.1 UTOFU_VERSION_*

uTofu翻訳時APIバージョン。

説明

これらのマクロはuTofu APIの翻訳時(ヘッダファイル)のバージョンを示します。

uTofu APIの実行時(ライブラリファイル)のバージョンはutofu_query_utofu_version()関数によって得られます。

マクロ

マクロ名	説明
UTOFU_VERSION_MAJOR	uTofu APIのmajorバージョン。
UTOFU_VERSION_MINOR	uTofu APIのminorバージョン。

第4章 uTofuの使用法

この章では、uTofuの使用法について説明します。

4.1 uTofuプログラムの設計

この節では、uTofuプログラムの設計にあたって必要な事項を説明します。

4.1.1 C言語以外からの利用

uTofuのC言語用のヘッダファイルutofu.hは、C++コンパイラによって処理された場合に、C言語へのリンケージ指定(extern "C")付きで関数などが宣言されるようになっています。そのため、C++ソースプログラムからヘッダファイルutofu.hを取り込むことにより、C言語のインターフェースを使用してC++プログラムでuTofuを使用できます。また、uTofuを使用したCオブジェクトプログラムやC++オブジェクトプログラムをFortranプログラムに言語間結合することにより、FortranプログラムでuTofuを間接的に使用することもできます。リンケージ指定の意味や言語間結合の方法については、C++の規格書や各コンパイラのマニュアルをお読みください。

4.1.2 MPIとの併用

注意

富士通MPIを使用する場合を前提として説明しています。ほかのMPIライブラリを使用する場合にはあてはまらないことがあります。

1つのプログラムの中でuTofuとMPIを併用できます。uTofuとMPIは論理的に独立してTofuインターコネクトによる通信を行うため、片方の通信中に他方が通信を行うことができます。

1つのプログラムの中でuTofuをMPIと併用する場合は、以下の制約があります。

- uTofuの関数の呼出しは、MPIのMPI_INITルーチンまたはMPI_INIT_THREADルーチンを呼び出してからMPI_FINALIZEルーチンを呼び出すまでの間で行う必要があります。これは、MPI_INITルーチンまたはMPI_INIT_THREADルーチンが呼ばれることで、計算ノード内のプロセス間やプロセス内のuTofuとMPIの間で通信資源(CQ)が適切に配分され、かつ、適切な通信経路が自動的に選択されるようになるためです。MPIのルーチンの詳細についてはMPIの規格書をお読みください。
- 複数のスレッドからuTofuの関数の呼出しとMPIのルーチンの呼出しが同時に行われる可能性がある場合、または、MPIのアシスタントコアを用いた非同期通信の促進の機能を使用する場合は、uTofuで作成するVCQはスレッド安全VCQまたはCQ専有VCQとして作成する必要があります。アシスタントコアを用いた非同期通信の促進の機能についてはMPIライブラリのマニュアルをお読みください。

1つのプログラムの中でuTofuをMPIと併用しない場合は、以下の制約があります。

- ある計算ノード内にMPIプロセスが存在する場合は、その計算ノード内のほかのプロセスでuTofuを使用できません。MPIプロセスとuTofuプロセスの間で通信資源が競合するためです。これは、MPIプロセスから生成された子プロセスでuTofuを使用できないことも意味します。

1つのプログラムの中でuTofuをMPIと併用しない場合は、以下の点に注意が必要です。

- uTofuによる通信を実行する前に通信相手プロセスにVCQ IDやVBG IDなどを通知する必要がありますが、MPI通信以外の手段を使って通知する必要があります。
- 1つの計算ノード内で複数のuTofuプロセスを実行する場合は、プロセス間で通信資源(CQ)が重複しないようにする必要があります。

uTofuプログラムを簡単に作成したい場合は、MPI通信が必要なくてもMPI_INITルーチンやMPI_INIT_THREADルーチンを呼び、MPI通信でVCQ IDやVBG IDなどを通知することをお勧めします。

なお、ここでは、MPI規格の表現に沿い、C言語、Fortran、C++の関数とFortranのサブルーチンをまとめてルーチンと呼び、MPIのルーチン名はすべて大文字で記載しています。

4.1.3 通信可能な範囲

“4.3.1 uTofuプロセスの生成”において説明するように、uTofuプログラムはジョブとして計算ノード上で実行します。あるジョブとして実行されたuTofuプロセスが通信できる範囲は、そのジョブの範囲です。これはMPIプロセスが通信できる範囲と同じです。

4.1.4 通信集中の回避

1つの計算ノードに同時に多数(数千から数万以上)の通信が集中すると、その計算ノード付近のTofuネットワークが混雑し、ネットワーク処理が著しく遅くなることがあります。デバイスドライバ(Tofuドライバ)によってそのような現象が検出されると、uTofu実装はエラーメッセージを出力してプログラムを強制的に終了させる場合があります。

例えば以下のような対策を行い、1つの計算ノードに同時に多数の通信が集中しないようにしてください。

- アルゴリズムを見直し、通信が特定の計算ノードに集中しないようにする。
- プロセス間の同期処理を挿入し、通信が時間的に分散するようにする。
- 1つの計算ノードに対して繰り返し通信する場合は、繰返しに間隔を設け、通信が時間的に分散するようにする。

4.2 uTofuプログラムの翻訳/結合

uTofuプログラムの翻訳/結合は、ほかのC言語のプログラムと同様に行うことができます。ただし、結合時にはコンパイラやリンカに-ltofucomオプションを渡して実行可能プログラムに共有ライブラリlibtofucom.soを結合する必要があります。

通常は、ヘッダファイルutofu.hや共有ライブラリlibtofucom.soが存在するディレクトリは、コンパイラやリンカの標準の検索パスリストに含まれます。そのため、-Iオプションや-Lオプションでディレクトリを指定する必要はありません。

uTofuプログラムの翻訳/結合には、ログインノードのクロスコンパイラまたは計算ノードのネイティブコンパイラをお使いください。1つのプログラムの中でuTofuとMPIを併用する場合は、MPIの翻訳/結合コマンドをお使いください。クロスコンパイラやネイティブコンパイラについては各コンパイラのマニュアルを、MPIの翻訳/結合コマンドについてはMPIライブラリのマニュアルをお読みください。

例

mpifccpxコマンドによるMPIプログラムの翻訳/結合例

1. 利用者プログラムtest.cを翻訳し、オブジェクトプログラムtest.oを作成します。

```
$ mpifccpx -c test.c
```

2. オブジェクトプログラムtest.oを結合編集して、実行可能プログラムtestを作成します。

```
$ mpifccpx -ltofucom -o test test.o
```

4.3 uTofuプログラムの実行

この節では、uTofuプログラムの実行にあたって必要な事項を説明します。

4.3.1 uTofuプロセスの生成

uTofuプログラムの実行は、計算ノード上で行う必要があります。計算ノード上での実行は、利用者が直接に行うのではなく、uTofuプログラムを実行するジョブの投入をジョブ運用ソフトウェアに依頼する形で行います。ジョブ投入方法については、ジョブ運用ソフトウェアのマニュアルをお読みください。

1つのプログラムの中でuTofuとMPIを併用する場合は、ジョブの中でmpiexecコマンドを使用してuTofuプログラムを実行します。“4.3.2 環境変数”において説明するuTofu固有の環境変数を必要に応じて設定できることを除いて、特別な準備は必要ありません。mpiexecコマンドの使用方法についてはMPIライブラリのマニュアルをお読みください。

1つのプログラムの中でuTofuとMPIを併用しない場合でも、ジョブの中でmpiexecコマンドを使用してuTofuプログラムを実行できます。生成されるuTofuプロセスの数と位置は、MPIプログラムの場合と同じです。富士通MPIのmpiexecコマンドの--debuglibオプションは無視されます。富士通MPIのmpiexecコマンドの--mcaオプションや-amオプションで指定できるMCAパラメーターのうち、MPIライブラリの動作だけに関するものは、無視されます。

1つのプログラムの中でuTofuとMPIを併用しない場合は、mpiexecコマンドを使用せずにuTofuプログラムを直接実行することもできます。

ジョブ種別がノード共有ジョブの場合は、uTofuを使用できません。ノード共有ジョブの詳細については、ジョブ運用ソフトウェアのマニュアルをお読みください。

4.3.2 環境変数

uTofuの一部の動作は、環境変数によって変更できます。uTofuの環境変数は、uTofuプロセスに対して、uTofuまたはMPIの関数が最初に呼ばれる前に、設定されている必要があります。uTofuまたはMPIの関数を1つでも呼んだ後にuTofuの環境変数を設定・変更・解除した場合は、動作が保証されません。

4.3.2.1 UTOFU_NUM_EXCLUSIVE_CQS

環境変数UTOFU_NUM_EXCLUSIVE_CQSによって、TNIあたりのCQ専有VCQ用のCQの要求数を設定できます。

0以上の数値を指定でき、省略値は0です。

MPIライブラリは、集団通信の高速化のために、TNIあたり複数のCQを使用することがあります。そのためのCQは予約されていますが、CQの数は有限のため、utofu_create_vcq()関数でCQ専有VCQを作成しようとしたときに、CQが不足して作成できないことがあります。

この環境変数でCQ専有VCQ用のCQの要求数を設定すると、できるだけその要求数のCQをCQ専有VCQ用に予約します。CQの数は有限であり、それを計算ノード内のすべてのプロセスに配分するため、要求数だけ予約できないこともあります。

uTofuとMPIを併用する場合は、1以上の値を設定すると、MPIライブラリの集団通信用のCQが減り、その性能に影響を与えることがあります。

CQ専有VCQについては、“3.3.4.1 UTOFU_VCQ_FLAG_*”のUTOFU_VCQ_FLAG_EXCLUSIVEマクロの説明を参照してください。

4.3.2.2 UTOFU_NUM_SESSION_MODE_CQS

環境変数UTOFU_NUM_SESSION_MODE_CQSによって、TNIあたりのセッションモードVCQ用のCQの要求数を設定できます。

0以上の数値を指定でき、省略値は3です。

MPIライブラリは、集団通信の高速化のために、TNIあたり複数のセッションモードCQを使用することがあります。CQの数は有限のため、uTofuプログラムがutofu_create_vcq()関数でセッションモードVCQを作成しようとしたときに、CQが不足して作成できないことがあります。

この環境変数でセッションモードVCQ用のCQの要求数を設定すると、MPIライブラリが使用する分も含めて、できるだけその要求数のCQをセッションモードVCQ用に予約します。CQの数は有限であり、それを計算ノード内のすべてのプロセスに配分するため、要求数だけ予約できないこともあります。

この環境変数と環境変数UTOFU_NUM_EXCLUSIVE_CQSの両方を指定し、CQの数が足りない場合は、環境変数UTOFU_NUM_EXCLUSIVE_CQSの指定の方が優先されます。

uTofuとMPIを併用する場合は、3未満の値を設定すると、MPIライブラリの集団通信用のCQが減り、その性能に影響を与えることがあります。

セッションモードVCQについては、“2.2.8 セッションモード”を参照してください。

4.3.2.3 UTOFU_NUM_MRQ_ENTRIES

環境変数UTOFU_NUM_MRQ_ENTRIESによって、フリーモードVCQのMRQのエントリ数を設定できます。

2048、8192、32768、131072、524288、または2097152のいずれかの数値を指定でき、省略値は131072です。これらの数値の代わりに、それぞれ、2Ki、8Ki、32Ki、128Ki、512Ki、または2Miのいずれかの文字列も指定できます。大文字・小文字は区別されません。それら以外の値を指定した場合は、一番近い値が指定されたものとみなされます。一番近い値が2つある場合、大きい方の値が指定されたものとみなされます。

Tofuインターコネクトでは、MRQにディスクリプタが書き込まれることによって、通信の完了が通知されます。utofu_poll_mrq()関数が呼ばれずにMRQにディスクリプタが溜まり続けると、MRQオーバーフローというエラーが発生します。MRQのエントリ数を多くすると、MRQオーバーフローの発生確率を減らすことができます。しかし、消費するメモリが多くなるというデメリットがあります。

4.3.2.4 UTOFU_NUM_MRQ_ENTRIES_SESSION

環境変数UTOFU_NUM_MRQ_ENTRIES_SESSIONによって、セッションモードVCQのMRQのエントリ数を設定できます。

指定できる値は環境変数UTOFU_NUM_MRQ_ENTRIESと同じです。

セッションモードVCQについては、“2.2.8 セッションモード”を参照してください。

4.3.2.5 UTOFU_SWAP_PROTECT

計算ノード上では、獲得済みのメモリ領域がメモリ不足等の理由でスワップアウトされることがあります。Tofuインターコネクトでのワンサイド通信に使用するメモリ領域がスワップアウトされると、そのメモリ領域を用いた通信でエラーが発生することがあります。この場合、ワンサイド

通信完了確認関数は以下のいずれかの復帰値を返します。utofu_poll_tcq()関数とutofu_poll_mrq()関数のどちらを呼び出したときにエラーの復帰値を返すかは、スワップアウトされるタイミング次第です。

- utofu_poll_tcq()関数の場合
  - UTOFU_ERR_TCQ_MEMORY
- utofu_poll_mrq()関数の場合
  - UTOFU_ERR_MRQ_LCL_MEMORY
  - UTOFU_ERR_MRQ_RMT_MEMORY

環境変数UTOFU_SWAP_PROTECTの指定により、通信用メモリ領域をスワップアウトの対象から除外するかどうかを選択できます。

整数値の0または1を指定でき、省略値は0です。0と1以外の値が指定された場合は、1が指定されたものと見なします。

本環境変数に1を指定した場合、uTofu実装内部でmlockシステムコールを呼ぶことで、Tofuインターコネクトでの通信に使用するメモリ領域がスワップアウトの対象にならないことを保証します。スワップアウトの対象から除外できるメモリ領域のサイズは、ソフト資源制限RLIMIT_MEMLOCKに従います。ジョブ実行時のRLIMIT_MEMLOCKの確認方法や変更方法については、ジョブ運用ソフトウェアのマニュアルをお読みください。

本環境変数に0を指定した場合、Tofuインターコネクトでの通信に使用するメモリ領域が、スワップアウトの対象になる可能性があります。

注意

通信用メモリ領域をスワップアウトの対象から除外すると、通信性能が低下することがあります。

RLIMIT_MEMLOCKの値が、ジョブで使用される通信用メモリ領域の合計サイズよりも小さい場合、本環境変数に1を指定してもスワップアウトによる通信エラーを回避できない場合があります。

第5章 uTofuの使用例

この章では、uTofuを使用したいくつかの例を示します。

この章で示すサンプルプログラムは、以下のように記述しています。

- “4.1.2 MPIとの併用”において説明したように、処理を単純にするため、MPIを併用しています。
- エラー発生有無の確認のために本来は関数の復帰値の確認が必要ですが、処理の流れに注目するため、最低限の確認しかしていません。
- 処理によって設定される値を説明するため、assert()関数形式マクロを使用しています。
- 変数名に使われているlclは自プロセス(local)、rmtは通信相手プロセス(remote)、numは数(number)を表します。

5.1 ワンサイド通信の使用例

5.1.1 Putによるping-pong通信の例

このプログラムでは、2つのプロセスが交互にPut通信を実行します。この通信パターンは一般にping-pong通信と呼ばれます。

それぞれのプロセスは、相手プロセスからPut通信によるデータの受信を確認した後に、相手プロセスへのPut通信を実行します。プログラムの前半では、リモートMRQ通知によって、データの受信を確認しています。プログラムの後半では、データが書き込まれるはずのメモリ領域をポーリングし、そのメモリ領域が更新されたことを確認することによって、データの受信を確認しています。

このプログラムは2プロセスで実行できます。

```
#include <stdlib.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

// 送信と送信完了確認
static void send(utofu_vcq_hdl_t vcq_hdl, utofu_vcq_id_t rmt_vcq_id,
                 utofu_stadd_t lcl_send_stadd, utofu_stadd_t rmt_recv_stadd, size_t length,
                 uint64_t edata, uintptr_t cbvalue, unsigned long int post_flags)
{
    int rc;

    // Put通信を指示
    utofu_put(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
              edata, post_flags, (void *)cbvalue);

    // TCQ通知を確認
    if (post_flags & UTOFU_ONESIDED_FLAG_TCQ_NOTICE) {
        void *cbdata;
        do {
            rc = utofu_poll_tcq(vcq_hdl, 0, &cbdata);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert((uintptr_t)cbdata == cbvalue);
    }

    // ローカルMRQ通知を確認
    if (post_flags & UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE) {
        struct utofu_mrq_notice notice;
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_PUT);
        assert(notice.edata == edata);
    }
```

```
}

// 受信確認
static void recv(utofu_vcq_hdl_t vcq_hdl, volatile uint64_t *recv_buffer, uint64_t expected_value,
                 uint64_t edata, unsigned long int post_flags)
{
    int rc;

    // リモートMRQ通知またはメモリ更新を確認
    if (post_flags & UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE) {
        struct utofu_mrq_notice notice;
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_RMT_PUT);
        assert(notice.edata == edata);
        assert(*recv_buffer == expected_value);
    } else {
        while (*recv_buffer != expected_value);
    }
}

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, lcl_rank, rmt_rank;
    unsigned long int post_flags;
    size_t num_tnis, length = sizeof(uint64_t);
    uint64_t edata, send_buffer;
    volatile uint64_t recv_buffer;
    uintptr_t cbvalue;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t lcl_vcq_id, rmt_vcq_id;
    utofu_stadd_t lcl_send_stadd, lcl_recv_stadd, rmt_recv_stadd;
    struct utofu_onesided_caps *onesided_caps;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes != 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &lcl_rank);
    rmt_rank = (lcl_rank == 0) ? 1 : 0;

    // ワンサイド通信に使用可能なTNIのIDを取得
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // ワンサイド通信に関するTNIの機能を問い合わせ
    utofu_query_onesided_caps(tni_id, &onesided_caps);

    // VCQを作成して対応するVCQ IDを取得
    utofu_create_vcq(tni_id, 0, &vcq_hdl);
    utofu_query_vcq_id(vcq_hdl, &lcl_vcq_id);
```

```
// VCQにメモリを登録して対応するSTADDを取得
utofu_reg_mem(vcq_hdl, (void *)&send_buffer, length, 0, &lcl_send_stadd);
utofu_reg_mem(vcq_hdl, (void *)&recv_buffer, length, 0, &lcl_recv_stadd);

// VCQ IDとSTADDを通信相手プロセスに通知
MPI_Sendrecv(&lcl_vcq_id, 1, MPI_UINT64_T, rmt_rank, 0,
             &rmt_vcq_id, 1, MPI_UINT64_T, rmt_rank, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(&lcl_recv_stadd, 1, MPI_UINT64_T, rmt_rank, 0,
             &rmt_recv_stadd, 1, MPI_UINT64_T, rmt_rank, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// 通知されたVCQ IDにデフォルト通信経路座標を埋め込み
utofu_set_vcq_id_path(&rmt_vcq_id, NULL);

recv_buffer = UINT64_MAX;
MPI_Barrier(MPI_COMM_WORLD);

// リモートMRQ通知を使ってping-pong通信
post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE |
             UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE |
             UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;

// iterationごとにedataの値を変えて、sendで実行したTOQに対応するrecvのMRQを識別
for (i = 0; i < iteration; i++) {
    // 使用できるedataは8bytesなので、256回ごとに0回にリセット
    edata = i % (1UL << (8 * onesided_caps->max_edata_size));
    cbvalue = i;
    send_buffer = i;
    if (lcl_rank == 0) {
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
             edata, cbvalue, post_flags);
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
        recv_buffer = UINT64_MAX;
    } else {
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
        recv_buffer = UINT64_MAX;
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
             edata, cbvalue, post_flags);
    }
}

recv_buffer = UINT64_MAX;
MPI_Barrier(MPI_COMM_WORLD);

// メモリ更新を使ってping-pong通信
post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE |
             UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;

// iterationごとにedataの値を変えて、sendで実行したTOQに対応するrecvのMRQを識別
for (i = 0; i < iteration; i++) {
    // 使用できるedataは8bytesなので、256回ごとに0回にリセット
    edata = i % (1UL << (8 * onesided_caps->max_edata_size));
    cbvalue = i;
    send_buffer = i;
    if (lcl_rank == 0) {
        send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
             edata, cbvalue, post_flags);
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
        recv_buffer = UINT64_MAX;
    } else {
        recv(vcq_hdl, &recv_buffer, send_buffer, edata, post_flags);
```

```
            recv_buffer = UINT64_MAX;
            send(vcq_hdl, rmt_vcq_id, lcl_send_stadd, rmt_recv_stadd, length,
                 edata, cbvalue, post_flags);
        }
    }

    // 資源を開放
    utofu_dereg_mem(vcq_hdl, lcl_send_stadd, 0);
    utofu_dereg_mem(vcq_hdl, lcl_recv_stadd, 0);
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.2 Getによる状況確認の例

このプログラムでは、ランク0が演算を行い、ランク1がその進捗状況を確認します。

ランク0は、演算の進捗状況に応じて随時、step変数を更新します。ランク1は、約5秒おきに、Get通信によってその変数の値を取得します。

コンパイラの最適化によってstep変数の更新処理が消えてしまわないように、step変数の宣言にvolatile修飾子を付ける必要があります。CPUによるstep変数のメモリ領域への書込みが不可分に行われることを前提にしています。

このプログラムは2プロセスで実行できます。

```
#define _POSIX_C_SOURCE 200809L // for clock_gettime

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <unistd.h>
#include <time.h>
#include <mpi.h>
#include <utofu.h>

static volatile unsigned long step;
static unsigned long sum;
static int loop_sec = 60;

// 演算処理
static void compute(void)
{
    struct timespec start_time, current_time;

    // 開始時刻を取得
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    do {
        // 演算
        for (int i = 0; i < 10000; i++) {
            sum += i;
        }

        // 進捗状況を示す値を更新
        step++;

        // 現在時刻を取得
        clock_gettime(CLOCK_MONOTONIC, &current_time);
    } while(current_time.tv_sec - start_time.tv_sec <= loop_sec);
}

// 状況確認処理
```

```
static void check(utofu_vcq_hdl_t vcq_hdl, utofu_vcq_id_t rmt_vcq_id, utofu_stadd_t lcl_stadd, utofu_stadd_t rmt_stadd)
{
    int rc;
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;
    void *cbdata;
    struct utofu_mrq_notice notice;
    struct timespec start_time, current_time;

    // 開始時刻を取得
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    do {
        sleep(5);

        // リモートプロセスの step 変数をローカルプロセスの step 変数に取得するGet通信を指示
        while (1) {
            rc = utofu_get(vcq_hdl, rmt_vcq_id, lcl_stadd, rmt_stadd, sizeof(step), 0, post_flags, NULL);
            if (rc != UTOFU_ERR_BUSY) {
                break;
            }
            utofu_poll_tcq(vcq_hdl, 0, &cbdata);
        }
        assert(rc == UTOFU_SUCCESS);

        // ローカルMRQ通知を確認
        do {
            rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
        } while (rc == UTOFU_ERR_NOT_FOUND);
        assert(rc == UTOFU_SUCCESS);
        assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_GET);

        // 取得した進捗状況の値を出力
        printf("step: %lu¥n", step);
        fflush(stdout);

        // 現在時刻を取得
        clock_gettime(CLOCK_MONOTONIC, &current_time);
    } while(current_time.tv_sec - start_time.tv_sec <= loop_sec);
}

int main(int argc, char *argv[])
{
    int rc, num_processes, rank;
    size_t num_tnis;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t lcl_vcq_id, rmt_vcq_id;
    utofu_stadd_t lcl_stadd, rmt_stadd;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes != 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // ワンサイド通信に使用可能なTNIのIDを取得
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
```

```
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // VCQを作成
    utofu_create_vcq(tni_id, 0, &vcq_hdl);

    // VCQにメモリを登録して対応するSTADDを取得
    utofu_reg_mem(vcq_hdl, (void *)&step, sizeof(step), 0, &lcl_stadd);

    if (rank == 0) {

        // VCQ IDを取得
        utofu_query_vcq_id(vcq_hdl, &lcl_vcq_id);

        // VCQ IDとSTADDをランク1に通知
        MPI_Send(&lcl_vcq_id, 1, MPI_UINT64_T, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&lcl_stadd,  1, MPI_UINT64_T, 1, 0, MPI_COMM_WORLD);

    } else {

        // VCQ IDとSTADDをランク0から通知
        MPI_Recv(&rmt_vcq_id, 1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(&rmt_stadd,  1, MPI_UINT64_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // 通知されたVCQ IDにデフォルト通信経路座標を埋め込み
        utofu_set_vcq_id_path(&rmt_vcq_id, NULL);

    }

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0) {
        compute();
    } else {
        check(vcq_hdl, rmt_vcq_id, lcl_stadd, rmt_stadd);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // 資源を開放
    utofu_dereg_mem(vcq_hdl, lcl_stadd, 0);
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.3 ARMWによるゲームの例

このプログラムでは、ランク0以外が一斉にランク0のnumber変数に1を加算する処理を繰り返し、運良くちょうど10000回目に加算したランクを勝者としています。

このプログラムは3プロセス以上およそ1000プロセス以下で実行できます。

注意

このプログラムのように1つのプロセスに多数の通信が集中すると、ネットワーク処理が著しく遅くなることがあります。このプログラムを数千プロセス以上で実行しないでください。詳細は“4.1.4 通信集中の回避”を参照してください。

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

int main(int argc, char *argv[])
{
    int rc, num_processes, rank;
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_LOCAL_MRQ_NOTICE;
    size_t num_tnis;
    uint64_t number = 1, lucky_number = 10000;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdl;
    utofu_vcq_id_t vcq_id;
    utofu_stadd_t stadd;
    void *cbdata;
    struct utofu_mrq_notice notice;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes < 2) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // ワンサイド通信に使用可能なTNIのIDを取得
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[0];
    free(tni_ids);

    // VCQを作成して対応するVCQ IDを取得
    utofu_create_vcq(tni_id, 0, &vcq_hdl);
    utofu_query_vcq_id(vcq_hdl, &vcq_id);

    // ランク0でVCQにメモリを登録して対応するSTADDを取得
    if (rank == 0) {
        utofu_reg_mem(vcq_hdl, &number, sizeof(number), 0, &stadd);
    }

    // VCQ IDとSTADDをランク0からランク0以外のプロセスに広報
    MPI_Bcast(&vcq_id, 1, MPI_UINT64_T, 0, MPI_COMM_WORLD);
    MPI_Bcast(&stadd,  1, MPI_UINT64_T, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank != 0) {
        // 広報されたVCQ IDにデフォルト通信経路座標を埋め込み
        utofu_set_vcq_id_path(&vcq_id, NULL);

        do {
            // ランク0の number 変数に1を加算するARMW通信を指示
            while (1) {
                rc = utofu_armw8(vcq_hdl, vcq_id, UTOFU_ARMW_OP_ADD, 1, stadd, 0, post_flags, NULL);
                if (rc != UTOFU_ERR_BUSY) {
                    break;
```

```
                }
                utofu_poll_tcq(vcq_hdl, 0, &cbdata);
            }
            assert(rc == UTOFU_SUCCESS);

            // ローカルMRQ通知を確認
            do {
                rc = utofu_poll_mrq(vcq_hdl, 0, &notice);
            } while (rc == UTOFU_ERR_NOT_FOUND);
            assert(rc == UTOFU_SUCCESS);
            assert(notice.notice_type == UTOFU_MRQ_TYPE_LCL_ARMW);

            // 加算前の値を確認
            if (notice.rmt_value == lucky_number) {
                printf("The winner is rank %d!¥n", rank);
            }
        } while (notice.rmt_value < lucky_number);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    // 資源を開放
    if (rank == 0) {
        utofu_dereg_mem(vcq_hdl, stadd, 0);
    }
    utofu_free_vcq(vcq_hdl);

    MPI_Finalize();

    return 0;
}
```

5.1.4 Putによるストライド通信の例

このプログラムでは、メモリ上でとびとびのデータを両隣のプロセスと相互に交換します。

複数のプロセスが並列に演算を繰り返し、次の段階の演算において、1つ前のランクと1つ後のランクの、前の段階での演算結果の一部を必要とするような処理を想定しています。この通信パターンは、一般に袖通信と呼ばれるものを、説明が単純になるように変形したものです。

通信時のTOQディスクリプタ作成のコストを削減するために、TOQディスクリプタ作成を最初に行い、それを使い回すようにしています。

メモリ上で連続しているデータの塊ごとにPut通信を行いますが、すべてのPut通信が完了したときにTCQ通知・リモートMRQ通知が行われるように、最後のPut通信だけでTCQ通知・リモートMRQ通知を行うようにしています。

演算と通信のタイミングを示すために、compute_center関数とcompute_edge関数を用意しています。しかし、演算内容は説明に関係ないため、それらの関数は実装していません。

このプログラムは3プロセス以上で実行できます。

```
#include <stdlib.h>
#include <mpi.h>
#include <utofu.h>

#define NX 100
#define NY 100

static void compute_center(double data[NY][NX])
{
    // 受信データを必要としない部分の演算
}

static void compute_edge(double data[NY][NX])
{
    // 受信データを必要とする部分の演算
```

```
}

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, lcl_rank, rmt_ranks[2];
    unsigned long int post_flags = UTOFU_ONESIDED_FLAG_TCQ_NOTICE | UTOFU_ONESIDED_FLAG_REMOTE_MRQ_NOTICE;
    size_t num_tnis, length, stride, toq_desc_sizes[2], toq_desc_size;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vcq_hdl_t vcq_hdls[2];
    utofu_vcq_id_t lcl_vcq_ids[2], rmt_vcq_ids[2];
    utofu_stadd_t lcl_stadds[2], rmt_stadds[2], lcl_offset, rmt_offset;
    double data[NY][NX];
    struct utofu_onesided_caps *onesided_caps[2];
    void *toq_descs[2], *cbdata;
    struct utofu_mrq_notice notice;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes < 3) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &lcl_rank);
    rmt_ranks[0] = (lcl_rank + num_processes - 1) % num_processes; // 1つ前のランク
    rmt_ranks[1] = (lcl_rank + 1) % num_processes;                 // 1つ後のランク

    // ワンサイド通信に使用可能なTNIのIDを取得
    rc = utofu_get_onesided_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    for (i = 0; i < 2; i++) {
        // 使用可能なTNIが2つ以上ある場合は、負荷分散のため2つの通信相手プロセスに異なるTNIを使用
        tni_id = (i == 0 || num_tnis == 1) ? tni_ids[0] : tni_ids[1];

        // ワンサイド通信に関するTNIの機能を問い合わせ
        utofu_query_onesided_caps(tni_id, &onesided_caps[i]);

        // VCQを2つ作成して対応するVCQ IDを取得
        utofu_create_vcq(tni_id, 0, &vcq_hdls[i]);
        utofu_query_vcq_id(vcq_hdls[i], &lcl_vcq_ids[i]);

        // VCQにメモリを登録して対応するSTADDを取得
        utofu_reg_mem(vcq_hdls[i], data, sizeof(data), 0, &lcl_stadds[i]);
    }
    free(tni_ids);

    // VCQ IDとSTADDを1つ前のランクと1つ後のランクに通知
    MPI_Sendrecv(&lcl_vcq_ids[0],  1, MPI_UINT64_T, rmt_ranks[0], 0,
                 &rmt_vcq_ids[1],  1, MPI_UINT64_T, rmt_ranks[1], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_vcq_ids[1],  1, MPI_UINT64_T, rmt_ranks[1], 0,
                 &rmt_vcq_ids[0],  1, MPI_UINT64_T, rmt_ranks[0], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_stadds[0],   1, MPI_UINT64_T, rmt_ranks[0], 0,
                 &rmt_stadds[1],   1, MPI_UINT64_T, rmt_ranks[1], 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&lcl_stadds[1],   1, MPI_UINT64_T, rmt_ranks[1], 0,
                 &rmt_stadds[0],   1, MPI_UINT64_T, rmt_ranks[0], 0,
```

```
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// 通知されたVCQ IDにデフォルト通信経路座標を埋め込み
utofu_set_vcq_id_path(&rmt_vcq_ids[0], NULL);
utofu_set_vcq_id_path(&rmt_vcq_ids[1], NULL);

// NX * NY の長方形のデータの左端を1つ前のランク、右端を1つ後のランクに送信するPut通信を準備
length = sizeof(double);
stride = length * NX;
for (i = 0; i < 2; i++) {
    toq_descs[i] = malloc(onesided_caps[i]->max_toq_desc_size * NY);
    // NY - 1 個のPut通信はTCQ通知・リモートMRQ通知なし
    lcl_offset = length * ((i == 0) ? 1 : (NX - 2));
    rmt_offset = length * ((i == 0) ? 0 : (NX - 1));
    utofu_prepare_put_stride(vcq_hdls[i], rmt_vcq_ids[i], lcl_stadds[i] + lcl_offset, rmt_stadds[i] + rmt_offset,
                             length, stride, NY - 1, 0, 0, toq_descs[i], &toq_desc_sizes[i]);
    // 最後の1個のPut通信はTCQ通知・リモートMRQ通知あり
    lcl_offset += stride * (NY - 1);
    rmt_offset += stride * (NY - 1);
    // TOQとMRQの対応関係を識別しない場合、edataは0でかまわない
    utofu_prepare_put(vcq_hdls[i], rmt_vcq_ids[i], lcl_stadds[i] + lcl_offset, rmt_stadds[i] + rmt_offset,
                      length, 0, post_flags, (char *)toq_descs[i] + toq_desc_sizes[i], &toq_desc_size);
    toq_desc_sizes[i] += toq_desc_size;
}

for (i = 0; i < iteration; i++) {
    // 準備していたPut通信を指示
    // (ループの1つ前の段階での演算結果を送信)
    utofu_post_toq(vcq_hdls[0], toq_descs[0], toq_desc_sizes[0], NULL);
    utofu_post_toq(vcq_hdls[1], toq_descs[1], toq_desc_sizes[1], NULL);

    // 最後のPut通信のTCQ通知(送信完了)を確認
    do {
        rc = utofu_poll_tcq(vcq_hdls[0], 0, &cbdata);
    } while (rc == UTOFU_ERR_NOT_FOUND);
    do {
        rc = utofu_poll_tcq(vcq_hdls[1], 0, &cbdata);
    } while (rc == UTOFU_ERR_NOT_FOUND);

    // 受信データを必要としない部分の演算
    // (送信が完了したため送信データ領域を更新しても良い)
    compute_center(data);

    // 最後のPut通信のリモートMRQ通知(受信完了)を確認
    do {
        rc = utofu_poll_mrq(vcq_hdls[0], 0, &notice);
    } while (rc == UTOFU_ERR_NOT_FOUND);
    do {
        rc = utofu_poll_mrq(vcq_hdls[1], 0, &notice);
    } while (rc == UTOFU_ERR_NOT_FOUND);

    // 受信データを必要とする部分の演算
    // (受信が完了したため受信データ領域を参照しても良い)
    compute_edge(data);
}

// 資源を開放
for (i = 0; i < 2; i++) {
    utofu_dereg_mem(vcq_hdls[i], lcl_stadds[i], 0);
    utofu_free_vcq(vcq_hdls[i]);
    free(toq_descs[i]);
}
```

```
    MPI_Finalize();

    return 0;
}
```

5.2 バリア通信の使用例

5.2.1 バリア同期とリダクション演算の例

このプログラムでは、プロセス間でバリア同期とリダクション演算を行います。リダクション演算では、すべてのプロセスの持つ浮動小数点データの和を計算します。

あるプロセスでutofu_poll_barrier()関数やutofu_poll_reduce_double()関数のループを抜けると、ほかのすべてのプロセスでutofu_barrier()関数やutofu_reduce_double()関数が呼ばれたことが保証されます。

リダクション演算ではすべてのプロセスで同じ計算結果が得られます。

このプログラムは8プロセスで実行できます。

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <mpi.h>
#include <utofu.h>

int main(int argc, char *argv[])
{
    int i, rc, iteration = 100, num_processes, rank;
    double data_in, data_out;
    size_t num_tnis;
    utofu_tni_id_t tni_id, *tni_ids;
    utofu_vbg_id_t lcl_vbg_ids[3], rmt_vbg_ids[8][3];
    struct utofu_vbg_setting vbg_settings[3];

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    if (num_processes != 8) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // バリア通信に使用可能なTNIのIDを取得
    rc = utofu_get_barrier_tnis(&tni_ids, &num_tnis);
    if (rc != UTOFU_SUCCESS || num_tnis == 0) {
        MPI_Abort(MPI_COMM_WORLD, 1);
        return 1;
    }
    tni_id = tni_ids[rank % num_tnis];
    free(tni_ids);

    // VBGを確保
    utofu_alloc_vbg(tni_id, 3, 0, lcl_vbg_ids);

    // 確保したVBGのIDの情報を共有
    MPI_Allgather(lcl_vbg_ids, 3, MPI_UINT64_T, rmt_vbg_ids, 3, MPI_UINT64_T, MPI_COMM_WORLD);

    // 8プロセスでのバタフライ交換アルゴリズムによるバリア回路を設定

    vbg_settings[0].vbg_id = lcl_vbg_ids[0];
```

```
    vbg_settings[0].src_lcl_vbg_id = lcl_vbg_ids[2];
    vbg_settings[0].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 4][2];
    vbg_settings[0].dst_lcl_vbg_id = lcl_vbg_ids[1];
    vbg_settings[0].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 1][1];
    vbg_settings[0].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    vbg_settings[1].vbg_id = lcl_vbg_ids[1];
    vbg_settings[1].src_lcl_vbg_id = lcl_vbg_ids[0];
    vbg_settings[1].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 1][0];
    vbg_settings[1].dst_lcl_vbg_id = lcl_vbg_ids[2];
    vbg_settings[1].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 2][2];
    vbg_settings[1].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    vbg_settings[2].vbg_id = lcl_vbg_ids[2];
    vbg_settings[2].src_lcl_vbg_id = lcl_vbg_ids[1];
    vbg_settings[2].src_rmt_vbg_id = rmt_vbg_ids[rank ^ 2][1];
    vbg_settings[2].dst_lcl_vbg_id = lcl_vbg_ids[0];
    vbg_settings[2].dst_rmt_vbg_id = rmt_vbg_ids[rank ^ 4][0];
    vbg_settings[2].dst_path_coords[0] = UTOFU_PATH_COORD_NULL;

    // VBGを設定
    utofu_set_vbg(vbg_settings, 3);

    // バリア通信を開始する前に、全プロセスでVBGが設定されるのをMPI_Barrierで待つ
    MPI_Barrier(MPI_COMM_WORLD);

    for (i = 0; i < iteration; i++) {

        // バリア同期を開始
        utofu_barrier(lcl_vbg_ids[0], 0);

        // バリア同期の完了を確認
        do {
            rc = utofu_poll_barrier(lcl_vbg_ids[0], 0);
        } while (rc == UTOFU_ERR_NOT_COMPLETED);
        assert(rc == UTOFU_SUCCESS);
    }

    for (i = 0; i < iteration; i++) {

        // リダクション演算を開始
        data_in = i + 0.1 * rank;
        utofu_reduce_double(lcl_vbg_ids[0], UTOFU_REDUCE_OP_BFPSUM, &data_in, 1, 0);

        // リダクション演算の完了を確認
        do {
            rc = utofu_poll_reduce_double(lcl_vbg_ids[0], 0, &data_out);
        } while (rc == UTOFU_ERR_NOT_COMPLETED);
        assert(rc == UTOFU_SUCCESS);
    }

    // 資源を開放する前に、全プロセスでバリア通信が完了するのをMPI_Barrierで待つ
    MPI_Barrier(MPI_COMM_WORLD);

    // 資源を開放
    utofu_free_vbg(lcl_vbg_ids, 3);

    MPI_Finalize();

    return 0;
}
```

第6章 システムの情報

この章では、システム固有の情報について説明します。

6.1 本システムの情報

この節では、本システム固有の情報について説明します。

6.1.1 本システムのバージョン情報

utofu_query_tofu_version()関数が返却する値を下表に示します。

表6.1 本システムのTofuインターコネクトのバージョン

種別	値
majorバージョン	3
minorバージョン	0

6.1.2 本システムの機能特性

ハードウェア資源の量を下表に示します。

表6.2 本システムのハードウェア資源の量

資源	値
計算ノードあたりのTNI数	6
TNIあたりの使用可能CQ数	9

仮想化された資源の量を下表に示します。

表6.3 本システムの仮想化された資源の量

資源	値
CQあたりの使用可能VCQ数	8
TNIあたりの使用可能VBG数 (始点・終点VBG数, 中継VBG数)	48 (16, 32)

utofu_query_onesided_caps()関数がutofu_onesided_caps構造体オブジェクトの各メンバに設定する値を下表に示します。各メンバの意味は、“3.2.2.1 struct utofu_onesided_caps”を参照してください。

表6.4 本システムのワンサイド通信の機能

メンバ名	値
flags	UTOFU_ONESIDED_CAP_FLAG_SESSION_MODE | UTOFU_ONESIDED_CAP_FLAG_ARMW
armw_ops	UTOFU_ONESIDED_CAP_ARMW_OP_CSWAP | UTOFU_ONESIDED_CAP_ARMW_OP_SWAP | UTOFU_ONESIDED_CAP_ARMW_OP_ADD | UTOFU_ONESIDED_CAP_ARMW_OP_XOR | UTOFU_ONESIDED_CAP_ARMW_OP_AND | UTOFU_ONESIDED_CAP_ARMW_OP_OR
num_cmp_ids	8
num_reserved_stags	256
cache_line_size	256
stag_address_alignment	256

メンバ名	値
max_toq_desc_size	64
max_putget_size	16,777,215 (2^{24} - 1)
max_piggyback_size	32
max_edata_size	1
max_mtu	1920
max_gap	255

utofu_query_barrier_caps()関数がutofu_barrier_caps構造体オブジェクトの各メンバに設定する値を下表に示します。各メンバの意味は、“3.2.2.2 struct utofu_barrier_caps”を参照してください。

表6.5 本システムのバリア通信の機能

メンバ名	値
flags	0
reduce_ops	UTOFU_BARRIER_CAP_REDUCE_OP_BARRIER | UTOFU_BARRIER_CAP_REDUCE_OP_BAND | UTOFU_BARRIER_CAP_REDUCE_OP_BOR | UTOFU_BARRIER_CAP_REDUCE_OP_BXOR | UTOFU_BARRIER_CAP_REDUCE_OP_MAX | UTOFU_BARRIER_CAP_REDUCE_OP_MAXLOC | UTOFU_BARRIER_CAP_REDUCE_OP_SUM | UTOFU_BARRIER_CAP_REDUCE_OP_BFPSUM
max_uint64_reduction	6
max_double_reduction	3

6.1.3 本システムの動作仕様

uTofuのインターフェース仕様に加え、以下の動作仕様があります。

6.1.3.1 ストロングオーダ・フラグ

ワンサイド通信でストロングオーダ・フラグを設定すると、先行する通信との間において主記憶の読込みや書込みの順序が保証されるだけでなく、1つのPut通信またはGet通信におけるCPUキャッシュライン間の読込みや書込みの順序も一部保証されます。具体的には、ローカル・リモートの計算ノードの主記憶における読込み元・書込み先の領域が複数のCPUキャッシュラインにまたがるときに、下表に示す内容が保証されます。

保証対象	条件	保証内容
読込み元末尾のCPUキャッシュライン	任意	ほかのCPUキャッシュラインからの読込みが完了した後に、末尾のCPUキャッシュラインからの読込みが開始される。
書込み先末尾のCPUキャッシュライン	読込み元における対応領域がページ境界をまたがらない場合	ほかのCPUキャッシュラインへの書込みが完了した後に、末尾のCPUキャッシュラインへの書込みが開始される。
	読込み元における対応領域がページ境界をまたがる場合	ほかのCPUキャッシュラインおよびページ境界の前方に対応する領域への書込みが完了した後に、ページ境界の後方に対応する領域への書込みが開始される。

これにより、末尾のCPUキャッシュラインの最終バイトをポーリングし、それが書き換わったことを確認することにより、全体の書込みが完了したことを確認できます。さらに、読込み元における書込み先末尾のCPUキャッシュラインに対応する領域がページ境界をまたがらないことをユーザーが保証できる場合は、キャッシュライン内の任意のバイトでポーリング可能です。

6.1.3.2 OSが管理するページサイズ

本システムにおいて、OSが管理するページサイズは以下の通りです。

- ラージページ機能使用時 : 2097152バイトまたは65536バイト
- ラージページ機能不使用時 : 65536バイト

“2.2.10 通信の完了確認と順序保証”に記載されているように、あるCPUキャッシュラインへの書込みがCPUキャッシュラインの粒度で行われることを保証するためには、そのCPUキャッシュラインに対応する読込み元領域が複数のページにまたがっていないことを確認する必要があります。ラージページ機能の使用の有無に関わらず、ある領域の先頭アドレスと“末尾アドレス −1”をそれぞれ65536で割り、両者の商が一致すれば、その領域は複数のページにまたがっていないと判断できます。また、本システムのCPUキャッシュラインサイズは256バイトのため、領域の先頭アドレスと“末尾アドレス −1”をそれぞれ256で割り、両者の商が一致すれば、その領域は複数のCPUキャッシュラインにまたがっていないと判断できます。

6.1.4 本システムにおける制約事項

6.1.4.1 uTofuプログラム内からのプロセス生成

システムコール(fork など)、ライブラリ関数(system など)、Fortran処理系のサービスルーチン(FORK など)などを使用して、uTofuプログラムから子プロセスの生成を行う場合、以下のような制約があります。

- プロセス生成のタイミングで、utofu_reg_mem()関数またはutofu_reg_mem_with_stag()関数によって登録され、かつ、対応するutofu_dereg_mem()関数によって解除されていないメモリ領域が存在する場合は、そのメモリ領域を含むページは子プロセスに引き継がれません。そのため、子プロセスはそのメモリ領域を含むページにアクセスできません。

同じメモリ領域を複数回または重なりのある複数のメモリ領域を登録した場合は、登録と同じ回数だけ解除しなければ、解除されていないことになります。

この制約のため、例えば、forkシステムコールによって生成された子プロセスが、execveシステムコールや_exitシステムコールを呼ぶまでの間に該当するメモリ領域に対してアクセスを行うと、子プロセスにおいてsegmentation faultのエラーが発生します。

アクセスの可否は、OSが管理するページの単位で判定されます。そのため、あるメモリ領域が登録中の場合には、その周辺のアドレスのメモリ領域にもアクセスできません。ラージページ機能を使用している場合は、ページの単位がラージページになります。ご注意ください。

6.1.4.2 Tofuインターコネクトのリンクダウン時に通信経路を変更する設定がされている場合の動作

uTofuプログラムを実行するジョブが、Tofuインターコネクトのリンクダウン時に通信経路を変更するよう設定されている場合は、uTofuプログラムの動作は不定となります。

Tofuインターコネクトのリンクダウンに対する動作の指定については、ジョブ運用ソフトウェアのマニュアルをお読みください。

第7章 エラーメッセージ

この章では、uTofu実装が出力するエラーメッセージについて説明します。

utofu: asynchronous error: *description* on TNI *tniid* BG *bgid*

- メッセージの説明

  バリア通信で内部エラーを検知しました。uTofuプログラムの実行を終了します。

- パラメーターの説明

  description: 検知した内部エラーに対応するメッセージ

  tniid: エラーが発生したTNIのID

  bgid: エラーが発生したBGのID

- 利用者の処置

  出力されたメッセージと合わせて担当保守員(SE)にご連絡ください。

utofu: asynchronous error: *description* on TNI *tniid* CQ *cqid*

- メッセージの説明

  ワンサイド通信で内部エラーを検知しました。uTofuプログラムの実行を終了します。

- パラメーターの説明

  description: 検知した内部エラーに対応するメッセージ

  tniid: エラーが発生したTNIのID

  cqid: エラーが発生したCQのID

- 利用者の処置

  ー ***description***が“MRQ Overflow”の場合

  MRQに書き込まれたMRQディスクリプタの数が、MRQのエントリ数を超えたことを表します。

  uTofuプログラムにおけるワンサイド通信の完了確認処理を見直してください。または、“4.3.2.3 UTOFU_NUM_MRQ_ENTRIES”および“4.3.2.4 UTOFU_NUM_MRQ_ENTRIES_SESSION”を参照してMRQのエントリ数を増やしてください。

  ー ***description***が“CQs Cacheflush Timeout”の場合

  Tofuインターコネクトにおいて輻輳が検知されたことを表します。

  uTofuプログラムの処理を見直して、特定の計算ノードに通信が集中しないようにしてください。詳細は“4.1.4 通信集中の回避”を参照してください。

  ー 上記以外の場合

  出力されたメッセージと合わせて担当保守員(SE)にご連絡ください。

用語集

ARMW(Atomic Read Modify Write)

ほかの計算ノードの主記憶において不可分な読出し、演算、書込みを行う、ワンサイド通信の機能です。TOQのARMWディスクリプタによって実行します。

BG(barrier gate)

TNIの資源の1つです。バリア回路を構成します。各BGはシグナルとパケットを1つずつ待ち合わせ、リダクション演算し、シグナルとパケットを1つずつ送出する機能を有します。始点・終点BGと中継BGの種類があります。

CQ(control queue)

ワンサイド通信においてソフトウェアからTNIを制御するためのキューです。TOQ、TCQ、およびMRQの3種類のキューで構成されます。TNIあたり複数のCQが存在します。

CQ専有VCQ(CQ-exclusive VCQ)

ほかのVCQとCQを共有しないVCQです。CQ専有VCQと別のVCQに対して、STADDの管理やワンサイド通信の実行を行う関数を、複数のスレッドから同時に呼び出すこともできます。異なるCQに対してそれらの関数を同時に呼び出した場合に、uTofu実装やハードウェアアクセスが並列に処理され、通信スループットが向上することがあります。VCQの作成時に`UTOFU_VCQ_FLAG_EXCLUSIVE`フラグを指定することで作成できます。

EDATA

TOQディスクリプタの書込み時に指定し、MRQディスクリプタの読込み時に返却される値です。TOQディスクリプタとMRQディスクリプタの対応関係を知るために使うことができます。

Get

ほかの計算ノードの主記憶から読出しを行う、ワンサイド通信の機能です。TOQのGetディスクリプタによって実行します。

MRQ(message receive queue)

CQあたり1つ存在する、主記憶上のキューです。ワンサイド通信においてデータ転送の完了通知が書き込まれます。環境変数`UTOFU_NUM_MRQ_ENTRIES`によって、エントリ数を指定できます。

MRQオーバーフロー(MRQ overflow)

MRQにローカル通知やリモート通知のMRQディスクリプタが書き込まれ続け、そのディスクリプタでMRQがオーバーフローする現象です。MRQディスクリプタが書き込まれ続けているのに`utofu_poll_mrq()`関数を呼ばないことが原因で発生します。

MTU(maximum transfer unit)

パケットの最大転送サイズです。ワンサイド通信において、MTUより大きいサイズのデータを送受信する場合は、TNIがデータをMTUごとに分割したパケットとして転送します。

NOP

何の通信も行わない、ワンサイド通信の機能です。TOQのNOPディスクリプタによって実行します。セッションモードでTOQディスクリプタの数合わせをするために使用します。

Put

ほかの計算ノードの主記憶に書込みを行う、ワンサイド通信の機能です。TOQのPutディスクリプタまたはPut Piggybackディスクリプタによって実行します。

STADD(steering address)

TNIが理解できるメモリアドレスです。ワンサイド通信では、TNIがメモリ(主記憶)を読み書きして計算ノード間でデータを転送します。しかし、ユーザープロセスから見える通常のメモリアドレスは、OSが割り当てた仮想アドレスであり、TNIは理解できません。そのため、ワ

ンサイド通信では、通信前にメモリの仮想アドレスをTNIが理解できるSTADDに変換して、そのSTADDをTNIに通知することによって、通信を行います。

STag(steering tag)

TNIが理解できるメモリ領域を示す整数値です。uTofuでは、通常、STADDの形で使用します。

TCQ(transmit complete queue)

CQあたり1つ存在する、主記憶上のキューです。ワンサイド通信においてデータ送出の完了通知に使用されます。同じCQのTOQと同じ数のエントリを有し、TOQとTCQのエントリは1対1で対応します。

TNI(Tofu network interface)

Tofuインターコネクトのネットワークインターフェースです。ソフトウェアからの指示にしたがってパケットを送受信し、ワンサイド通信とバリア通信を実行します。計算ノードあたり1つ以上のTNIが存在し、主記憶とTNRに接続されます。

TNR(Tofu network router)

Tofuインターコネクトのネットワークルータです。TNIから送出されたパケットを中継して宛先のTNIまで届けます。計算ノードあたり1つのTNRが存在し、TNIとTofuネットワークに接続されます。

Tofuインターコネクト(Tofu interconnect)

ハイエンドテクニカルコンピューティングサーバMP10システム、スーパーコンピュータPRIMEHPC FX10システム、スーパーコンピュータPRIMEHPC FX100システム、および本システムで使用されている計算ノード間インターコネクトの総称です。TNI、TNR、およびTofuネットワークから構成されます。単にTofuとも呼びます。

Tofuネットワーク(Tofu network)

Tofuインターコネクトのネットワークです。6次元の座標空間を持ちます。TNRを相互接続することによって構成されます。

TOQ(transmit order queue)

CQあたり1つ存在する、主記憶上のキューです。ワンサイド通信において通信の指示を出すために使用されます。

uTofu

ユーザー空間のプロセスがTofuインターコネクトを使用して通信を行うためのプログラミングインターフェースです。

この名称は、user-level Tofu communication interfaceに由来します。

VBG(virtual barrier gate)

BGをソフトウェア的に抽象化したものです。Tofuネットワーク内でVBGを一意に識別する64ビットの整数値をVBG IDと呼びます。

VCQ(virtual control queue)

CQをソフトウェア的に抽象化・仮想化したものです。uTofuでは1つのCQを複数のソフトウェアコンポーネントで共有するため、VCQを使用してCQを仮想化しています。Tofuネットワーク内でVCQを一意に識別する64ビット符号なし整数値をVCQ IDと呼びます。あるプロセス内でそのプロセスが作成したVCQをuTofuが扱うためのハンドルをVCQハンドル(VCQ handle)と呼びます。

計算ノード(compute node)

CPU、主記憶、TNI、およびTNRを備えたハードウェアの単位です。通常、OS(ホストOS)が動作する単位でもあります。

計算ノード座標(compute node coordinates)

Tofuネットワーク上で計算ノードの位置を示すX、Y、Z、A、B、Cの6次元の座標です。

コールバックデータ(callback data)

TOQディスクリプタの書込み時に指定し、TCQディスクリプタの読込み時に返却される値です。TOQディスクリプタとTCQディスクリプタの対応関係を知るために使うことができます。

コンポーネントID(component ID)

CQを共有する複数の上位コンポーネントのVCQを区別するために使われる値です。

シグナル(signal)

バリア通信において同一のTNIのBG間で伝達されるデータです。

主記憶(main memory)

計算ノードに存在し、CPUのロード・ストア命令などやTofuインターコネクトのワンサイド通信が読込み・書込みを行う記憶領域です。本書では単にメモリとも呼びます。

スレッド安全VBG(thread-safe VBG)

複数のスレッドから同時に使用できるVBGです。1つのスレッド安全VBGに対して、バリア通信の実行関数を、複数のスレッドから同時に呼び出すことができます。VBGの確保時に`UTOFU_VBG_FLAG_THREAD_SAFE`フラグを指定することで確保できます。

スレッド安全VCQ(thread-safe VCQ)

複数のスレッドから同時に使用できるVCQです。1つのスレッド安全VCQに対して、STADDの管理やワンサイド通信の実行を行う関数を、複数のスレッドから同時に呼び出すことができます。また、スレッド安全VCQと別のVCQに対して、それらの関数を、複数のスレッドから同時に呼び出すこともできます。VCQの作成時に`UTOFU_VCQ_FLAG_THREAD_SAFE`フラグを指定することで作成できます。

セッションモード(session mode)

CQの特殊なモードです。ワンサイド通信開始関数またはワンサイド通信一括開始関数を呼び出すと、通信指示のディスクリプタがTOQに書き込まれますが、TNIは通信を開始しません。さらにほかのPut通信が届くことで、TNIが通信を開始します。セッションモードのCQに作成されたVCQをセッションモードVCQと呼びます。

送信間隔(transmission gap)

パケットを送信するときのパケット間の間隔です。Tofuネットワーク内で通信経路の衝突が起きることが事前に分かっている場合、転送データの送出帯域を抑制することで輻輳による実効帯域の低下を緩和できる場合があります。ワンサイド通信において、送信間隔に0以外の値を指定すると、TNIがパケットを送信した後に次のパケットをすぐに送信せず、送出帯域を抑制できます。

通信経路(communication path)

ある計算ノードから別の計算ノードに対して通信を行うときに、Tofuネットワーク上でパケットが通る経路です。

通信経路ID(communication path ID)

ワンサイド通信を実行するときに通信経路を指定するために使用する値です。`utofu_get_path_id()`関数によって作成し、`UTOFU_ONESIDED_FLAG_PATH`関数形式マクロを通してワンサイド通信開始関数やワンサイド通信準備関数に指定します。

通信経路座標(communication path coordinates)

ある計算ノードから別の計算ノードに対して通信を行うときのTofuネットワーク上での通信経路を示すA、B、Cの3次元の座標です。

ディスクリプタ(descriptor)

データ転送の指示または完了通知を示す制御データです。TOQディスクリプタ、TCQディスクリプタ、およびMRQディスクリプタの3種類の系統が存在します。TOQディスクリプタは、ソフトウェアがTofuインターコネクトにデータ転送の指示を出すためにTOQに書き込み、Tofuインターコネクトが読み込んでその指示を実行します。TCQディスクリプタとMRQディスクリプタは、Tofuインターコネクトがデータ転送の完了をソフトウェアに通知するためにTCQまたはMRQに書き込み、ソフトウェアが読み込んで完了を確認します。TOQであればPutディスクリプタやGetディスクリプタなど、指示や完了通知の内容によってさらに細分化されます。

デフォルト通信経路(default communication path)

ワンサイド通信開始関数やワンサイド通信準備関数で通信経路IDを指定しなかった場合に使用される通信経路です。`utofu_set_vcq_id_path()`関数によってVCQ IDに通信経路座標を埋め込むことで使用します。

パケット(packet)

Tofuネットワーク内での通信データの転送単位です。ワンサイド通信では、サイズの大きいデータを送受信するときに、TNIがそのデータをパケットに分割して転送します。バリア通信では、同一または異なるTNIのBG間でパケットを送受信してデータを伝達します。

バリア通信(barrier communication)

計算ノード間のバリア同期を行う通信機能です。リダクション演算を同時に実行可能です。プロトコル処理は完全にオフロードされており、CPUによる制御を必要としません。

バリア同期(barrier synchronization)

バリア通信を使用したBG間の同期処理です。バリア通信に参加するBGをあらかじめ設定しておき、それぞれの始点・終点BGでバリア通信を開始します。バリア通信に参加するすべての始点・終点BGにおいてバリア通信が開始されると、それぞれの始点・終点BGでバリア通信が完了します。

ピギーバック(piggyback)

Putで低遅延の通信を行うための機構です。Putでは、通常、ソフトウェアがTOQディスクリプタを通じてTNIに送信指示を出し、TNIが主記憶からデータを読み出します。ピギーバックを使用すると、ソフトウェアがデータをTOQディスクリプタに埋め込むため、TNIが主記憶からデータを読み出す必要がなくなり、低遅延な通信を行うことができます。

富士通MPI(Fujitsu MPI)

Technical Computing Suiteに含まれるMPIライブラリです。

フリーモード(free mode)

CQの通常のモードです。ワンサイド通信開始関数またはワンサイド通信一括開始関数を呼び出すと、通信指示のディスクリプタがTOQに書き込まれ、TNIが通信を開始します。フリーモードのCQに作成されたVCQをフリーモードVCQと呼びます。

リダクション演算(reduction operation)

複数の値から1つの値を導出する演算です。

リモート(remote)

ワンサイド通信によって2つのVCQの間で通信を行うときの、通信の標的となる側のVCQです。またはそのVCQ側の計算ノード、プロセス、TNI、STADDなどです。逆はローカルです。

ローカル(local)

ワンサイド通信によって2つのVCQの間で通信を行うときの、通信の指示を出す側のVCQです。またはそのVCQ側の計算ノード、プロセス、TNI、STADDなどです。逆はリモートです。

ワンサイド通信(one-sided communication)

ほかの計算ノードの主記憶を参照・更新する通信機能です。Put、Get、ARMW、およびNOPの機能を持ちます。この通信方式は一般にRDMA(remote direct memory access)と呼ばれます。プロトコル処理は完全にオフロードされており、CPUによる制御を必要としません。