

# PGAS向け 低水準通信レイヤーの マルチスレッド実装

東京大学 大学院 情報理工学系研究科 電子情報学専攻

遠藤 亘, 田浦 健次郎

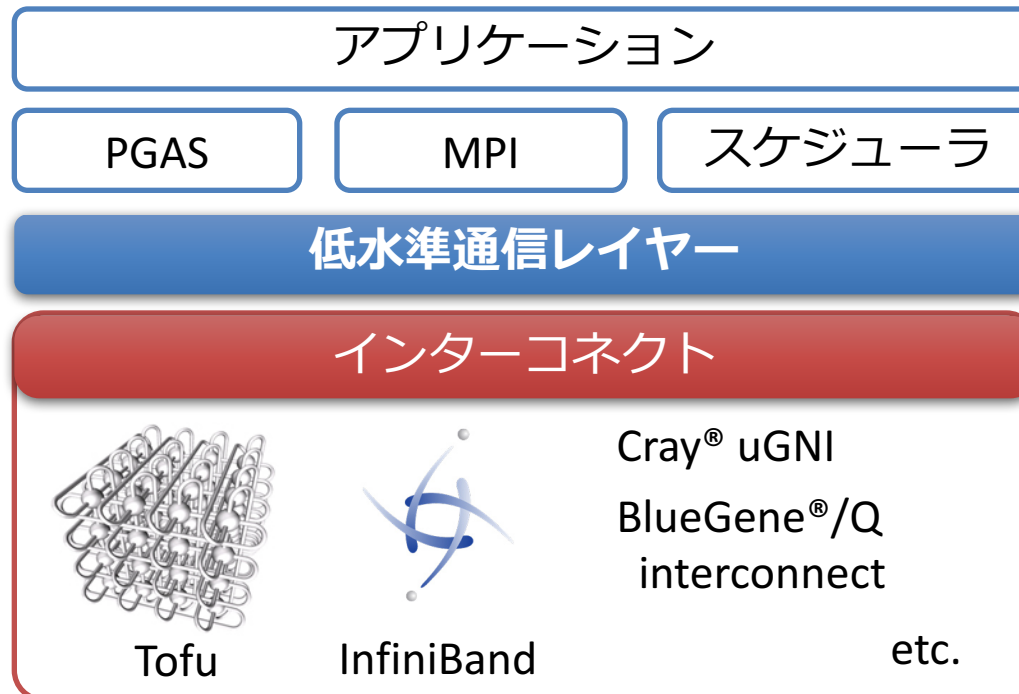
2016/8/9

SWoPP '16

# 低水準通信レイヤー

2

- インターコネクトの機能を抽象化
  - ハードウェア更新毎にシステム全体を再設計するのは非生産的
- 「システムのためのシステム」



# 本研究の貢献

3

- 低水準通信レイヤーのAPIを再定義
  - マルチスレッド対応 & 低オーバーヘッド
- Tofu用実装の性能
  - スレッドセーフティ確保のためのレイテンシ増加を19%に抑えた
  - 15スレッド時のメッセージレート低下を12%に抑えた
- InfiniBand用実装の性能
  - 自動的な通信集約によって、メッセージレートを向上させた

# 既存の低水準通信レイヤー


4

- 古くからあるもの（2002年頃～）
  - GASNet [Bonachea et al. '02], ARMCI [Nieplocha et al. '06]
- 近年登場したもの（2014年頃～）
  - libfabric [Grun et al. '15], UCX [Shamis et al. '15], ComEx [Daily et al. '14]
- ソフトウェア一般の評価指標
  - 移植性, APIの網羅性
- 通信システムの性能評価指標
  - レイテンシ, メッセージレート, (バンド幅)
  - **マルチスレッド性能**

既存研究であまり重視されていない

# 既存処理系の現状（例. GASNet）

5

- GASNetが提供する機能
  - **Remote Memory Access (RMA)**  PGASにとって最も重要
    - 他ノード上のメモリを読み書き
    - RDMAの抽象化
  - Active Messages (AM)
    - 他ノード上で関数を実行
  - 集団通信 (Collective Communication)
    - 全ノードが同期して通信
- GASNetの問題点
  - リモートアトミックのような新しい機能が欠落
  - マルチスレッド対応には粗粒度ロックを使う

# “ポストムーア時代”のハードウェア動向

6

	これからの予想	通信システムとして
コアあたり 計算能力	伸びない	<b>ソフトウェア オーバーヘッドの低減</b>
バンド幅	今後も向上見込み (例. OCS)	他のボトルネック解消 のために活用
レイテンシ	あまり縮まない	<b>レイテンシ隠蔽 が重要</b>
ノードあたり 計算能力	今後も増大	<b>マルチスレッド化 が重要</b>
ノード間 通信資源	ノード内コアが共有	

# レイテンシ隠蔽とマルチスレッド

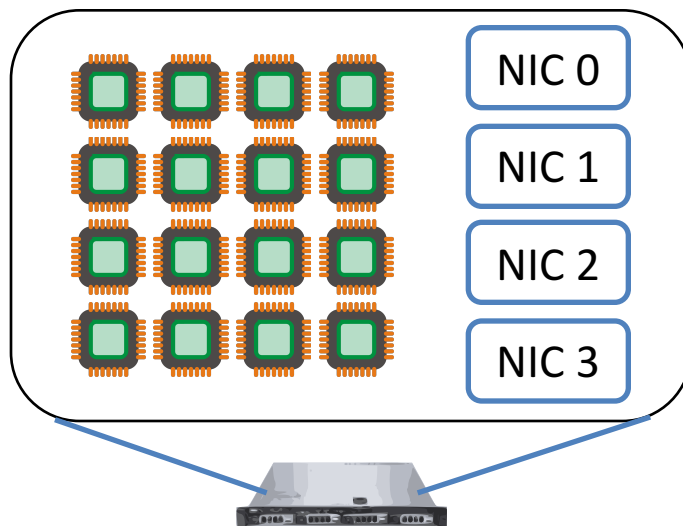
7

- レイテンシはあまり減らなくなった
  - (対照的に) バンド幅は増え続けている
- **マルチスレッドによるレイテンシ隠蔽**
  - 計算を進められるスレッドが先に進行する  
→ 計算資源を有効に使う
- 全スレッドが自由に通信を発行できるモデル
  - MPIだと“MPI\_THREAD\_MULTIPLE”
  - 生産性が高く, レイテンシにも強い

# 通信とマルチスレッド

8

- 1ノードあたりのコア数が増加
- 1ノードあたりの通信資源は増加していない
  - 理想的には「1コア1通信資源」
- ノード内のコア同士が通信資源を共有する必要



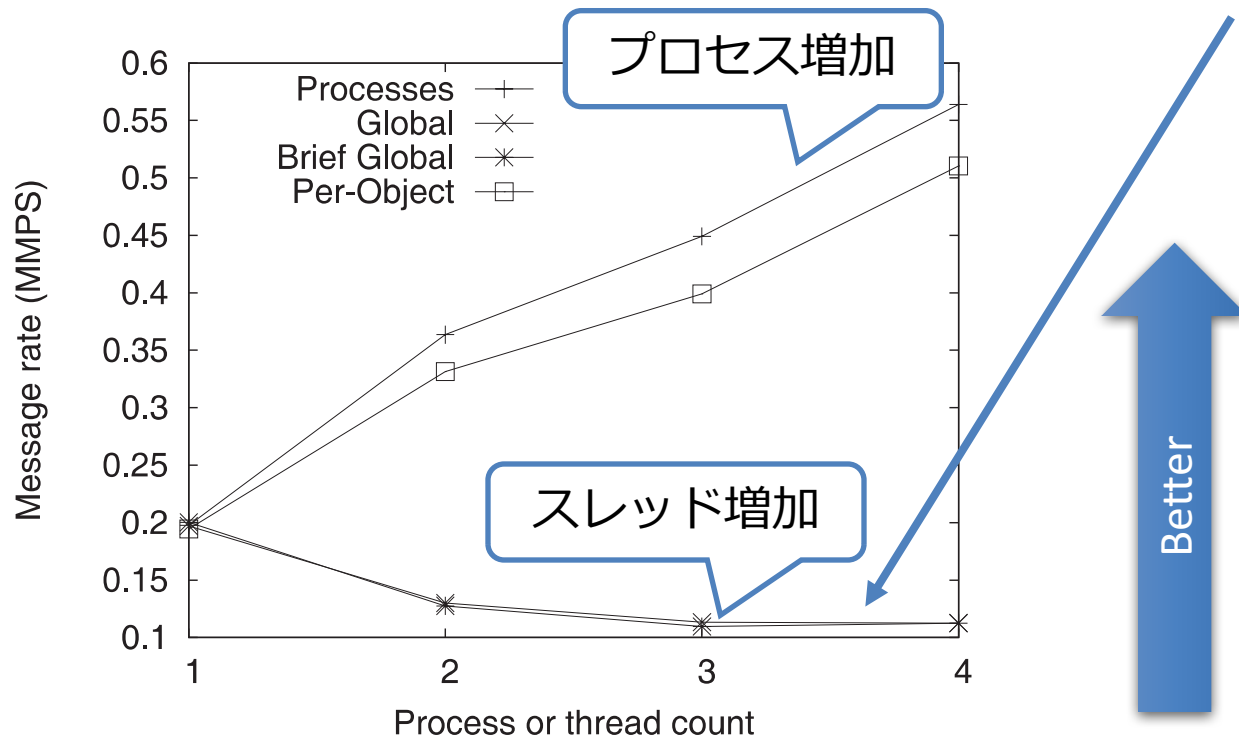
TofuのNICとCPU数の関係



# MPIとマルチスレッド

9

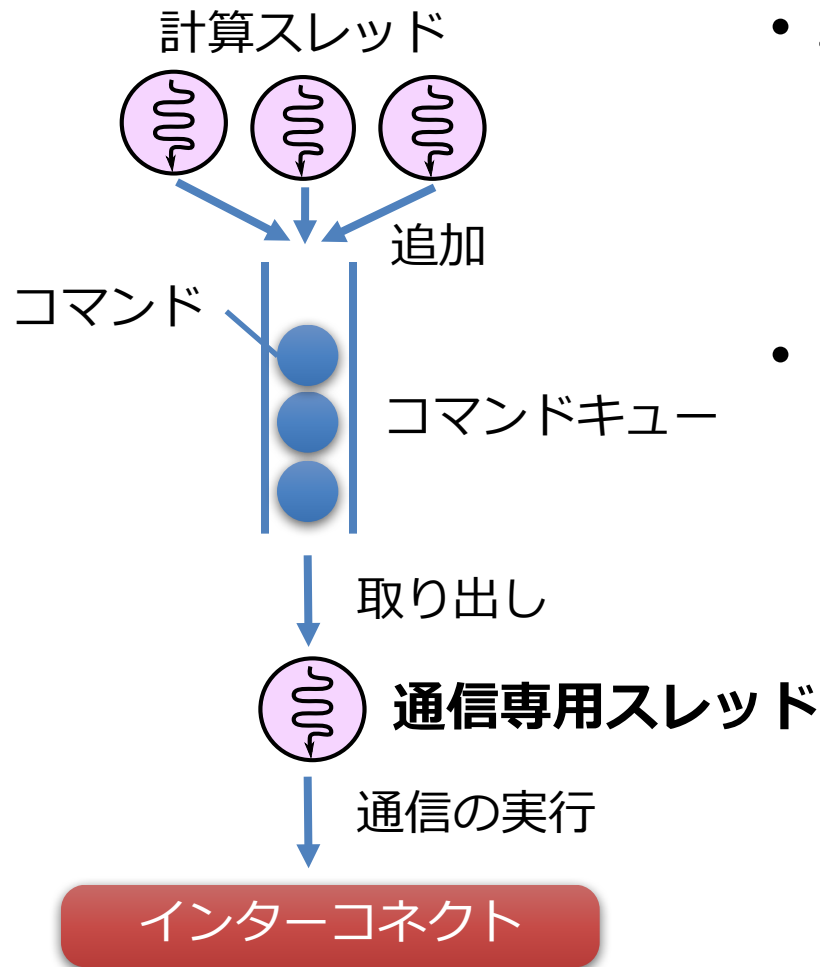
- MPICHのマルチスレッド性能 [Balaji et al. '10]
  - 粗粒度ロックによるスレッドセーフティの確保
  - スレッド数増加とともにメッセージレートが低下



MPICHにおけるMPI\_Sendのメッセージレート [Balaji et al. '10]

# Software Offloading [Vaidyanathan et al. '15]

10



- Software Offloadingとは？
  - **通信専用スレッド**を用意
  - 他スレッドはそのスレッドに通信処理を移譲する
- どのように移譲するか？
  - (MPIの) 通信要求を「**コマンドキュー**」に溜める

# オフローディングの利点・欠点

11

## ・利点

- ・ **レイテンシ隠蔽**が可能
  - ・ 計算スレッドは、キューに挿入さえすれば通信完了を待たなくてよい
- ・ (ロックに比べて) **並行性**が改善
  - ・ キューの実装次第
- ・ 通信を自動的に**集約**できる

## ・欠点

- ・ 通信1回分の処理が増えるため**レイテンシ増大**
- ・ 次の2択を迫られている
  - ・ 通信資源をめぐる衝突を回避するために、他コアに通信を移譲（オフロード）する
  - ・ レイテンシ削減のために、自コアから通信を発行することにこだわる

# オフローディングと通信ハードウェア

12

- ノード間の通信資源
  - **これからもノード内で共有されると予想**
    - 潤沢なコア数を活かしてオフローディングを行うべき
  - 各コアが独立したNICを扱えるハードウェアは現時点では存在しない
    - Intel® Omni-Path Architectureでは160コンテキストまでスケールする？
- ノード内通信も決して高速ではない
  - ノード内コア間通信も極力減らす

# オフローディングを行うレイヤー

13

- MPI
  - 例. Vaidyanathanらの研究
    - MPI内部を本格的にマルチスレッド化してはいない
- PGAS
  - 例. **ACP** [佐賀 et. al '15]
    - Tofu用の実装でオフローディングを行っている
- **低水準通信レイヤー**
  - 本研究がおそらく初

どのレイヤーで  
オフローディングを  
行うべきか？



# 提案手法

14

- 低水準通信レイヤーでのオフローディングを行う利点
  - システムの**直交性向上**
    - 並行性バグの削減
  - 複数システムの**連携**が容易
  - **各インターコネクトに合わせた**  
スケジューリング戦略を実装することが容易
    - 例. InfiniBandにおける自動的な通信集約

# 低水準通信レイヤーの要件

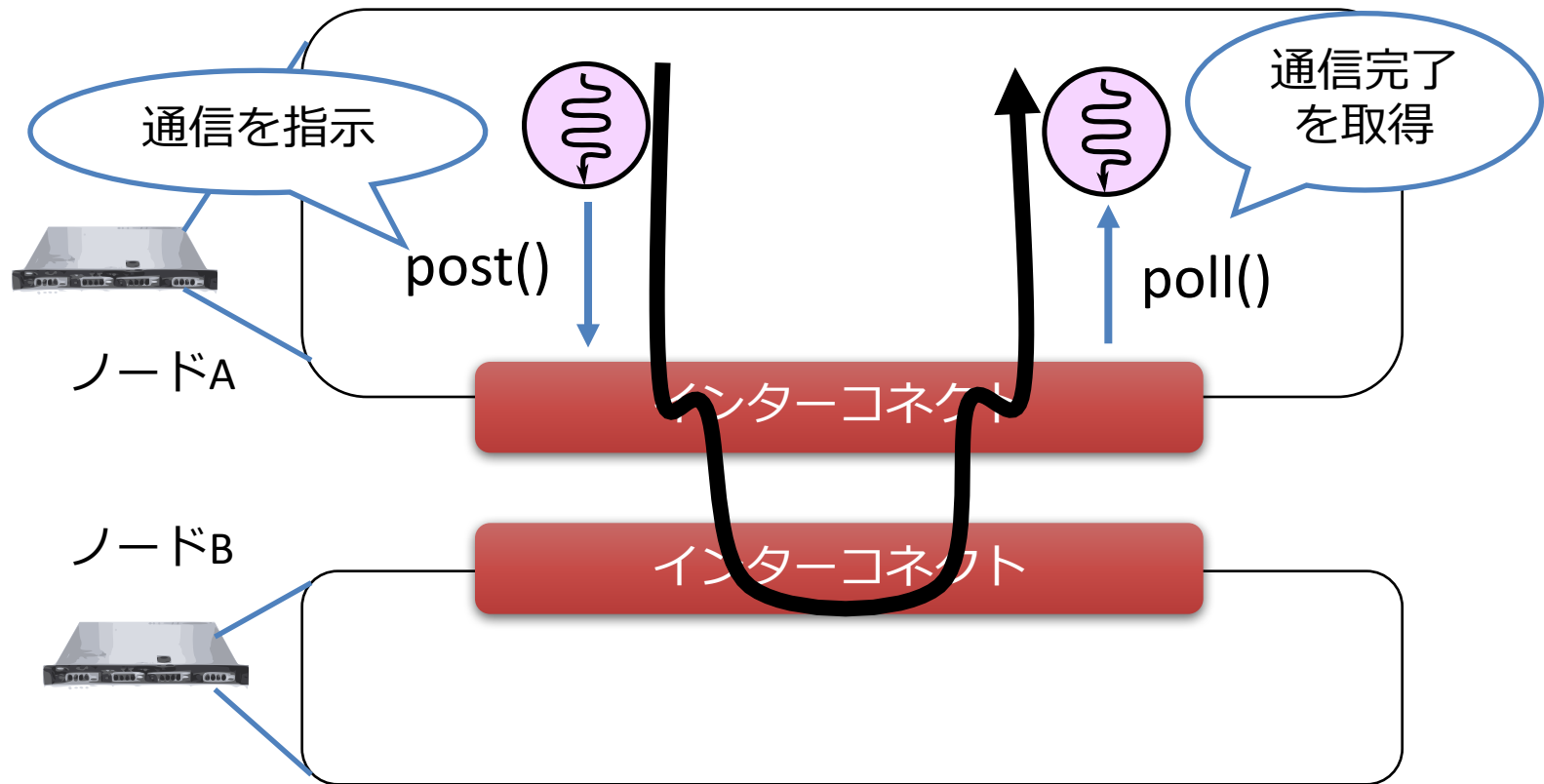
15

- 移植可能性
  - 他システムに移植しても性能が劣化しない
- **現実のハードウェアに近いAPI**
  - 不必要に抽象度を上げ過ぎない
- **マルチスレッド性能**
  - 複数スレッドが任意時点で通信を要求しても効率的
- **ノンブロッキングAPI**
  - レイテンシ隠蔽を低コストで行う

# インターコネクトAPI

16

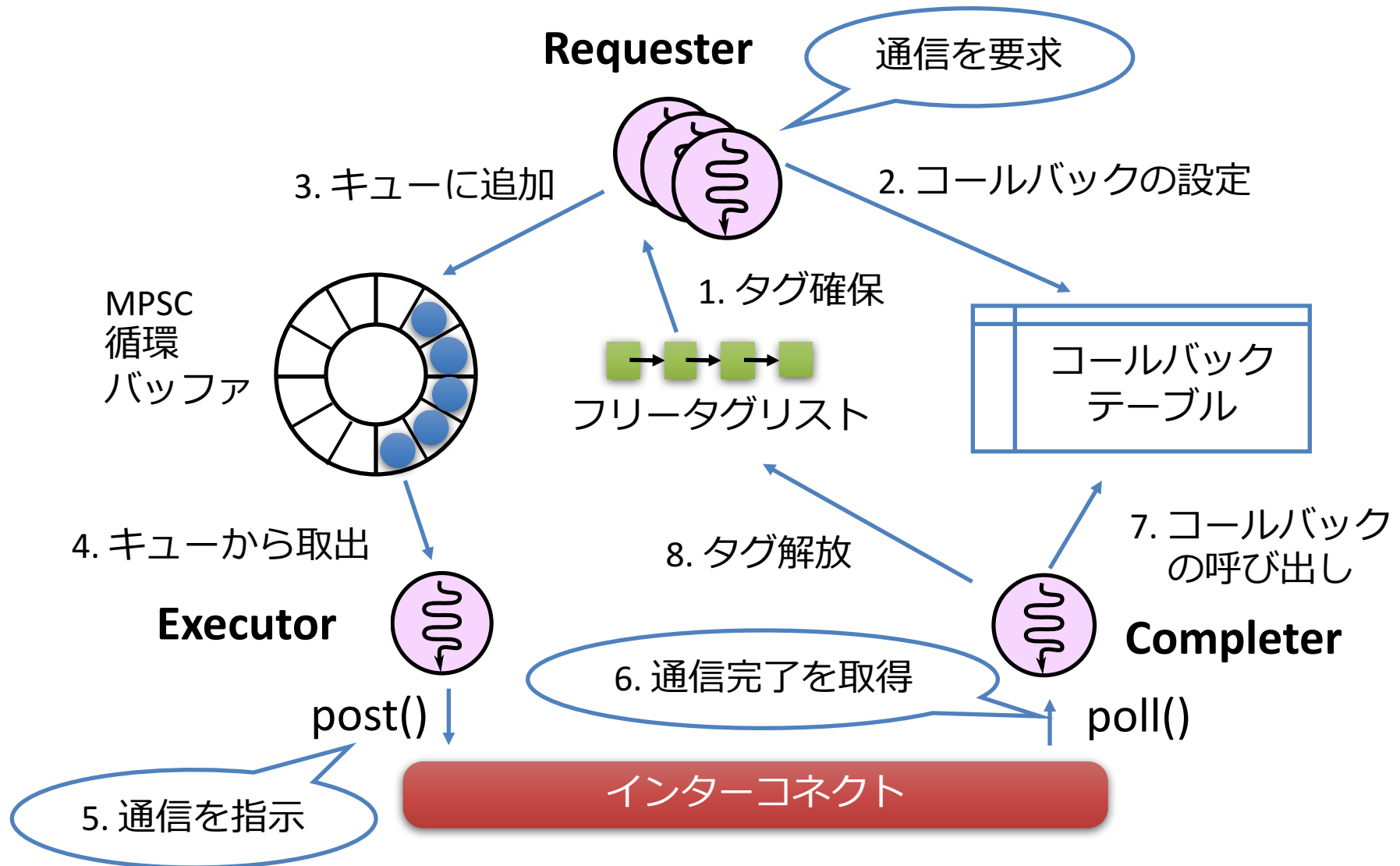
- 典型的なRDMA API
  - post して poll





# 提案システムの設計

17



# ノンブロッキングキュー

18

- キューの実装→マルチスレッド性能に大きく影響
    - A. 任意長キュー
      - 動的アロケーション→アトミック操作
    - B. **固定長キュー（循環バッファ）**
      - アトミック操作 = アロケーション
      - 実装が容易かつ高速
- 本研究で採用
- RDMAの最小レイテンシは数千クロック程度
    - オーバーヘッドとなる要因は全て排除すべき

資料中の「ロックフリー」は誤り  
(正しくはObstruction-freeまたはノンブロッキング)

# 固定長バッファの問題点

19

- キューが満杯になると追加できなくなる
  - [選択肢1] すぐさま再試行する
  - [選択肢2] 並行してできる他の仕事に切り替える
- どちらがよいかは通信システムだけでは判断不能
  - スケジューリングに関する情報が不可欠
- 通信が失敗したという情報だけを返して速やかに復帰するようAPIを定義

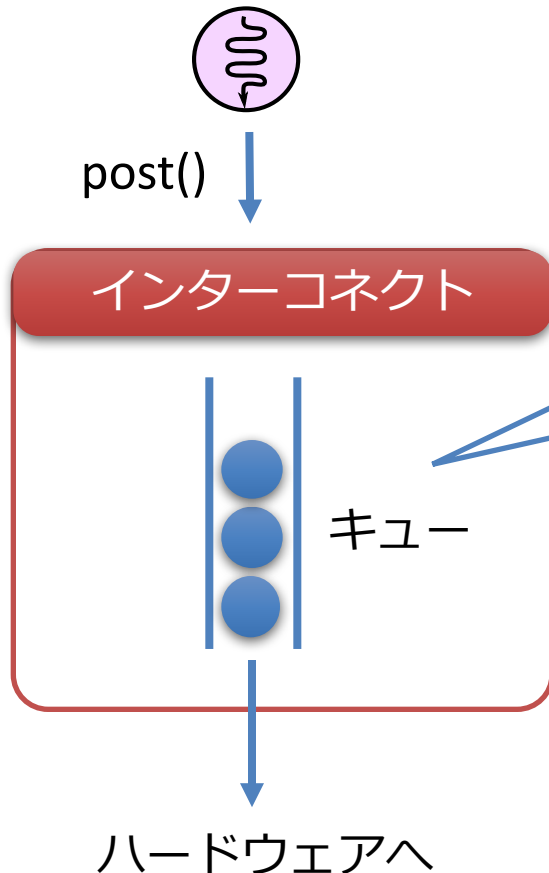
```
bool try_...(/**/);
```

- 「通信はもう処理しきれない」ということを伝える
- すぐに再試行するかどうか, yieldを入れるかどうかなどは上位層に委ねる

# “失敗可能”であること

20

- インターコネクトAPIの内部にもキューがある



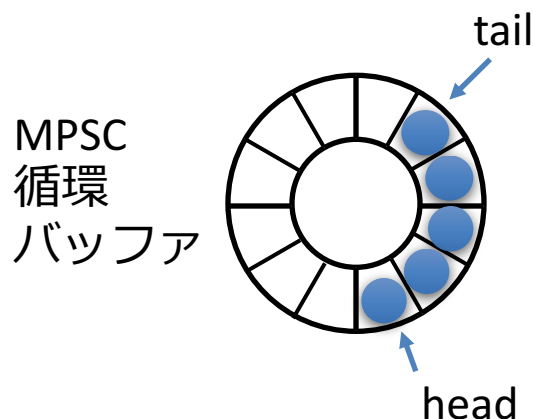
これを直接複数スレッドから  
並行操作できるのが理想だが  
現実には難しい

- 内部キューのサイズは固定長
  - 満杯になるとやはりpost時に「失敗」する
  - 「無限に通信を発行できる」という仮定が不自然

# ノンブロッキング循環バッファの実装

21

- **Multiple-Producer Single-Consumer (MPSC)**
  - Requester (=Producer) は複数
  - Executor (=Consumer) は1スレッド
- head/tailのカウンタで管理
- tailが進む ≠ Producerによって値が書き込まれる
  - Consumerからの可視性を制御するためのフラグを用いる



# Requester

22

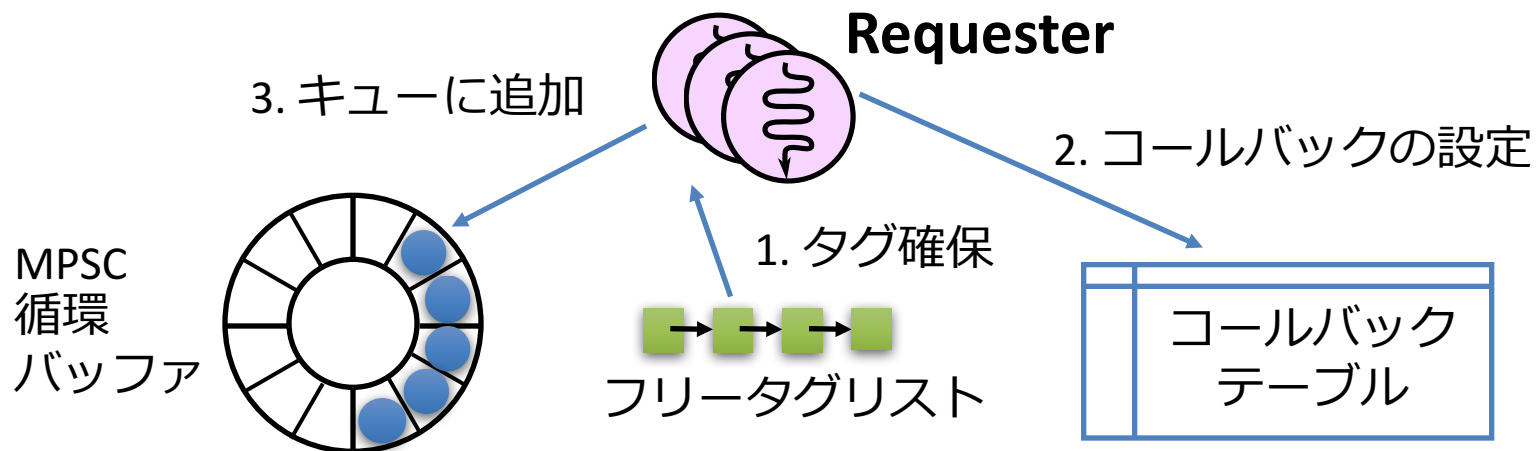
- キューに通信要求を投入する役割 (Producer)

```
bool try_read_async(/*...*/) {  
    if (/*タグが確保できない*/) return false;  
    コールバックの設定;  
    do { if (/*キューが満杯*/) return false; }  
    while (! CAS(/*tailを進める*/));  
    コマンドの設定;  
    可視フラグをオンにする;  
    return true;  
}
```

アセンブリで  
数十行



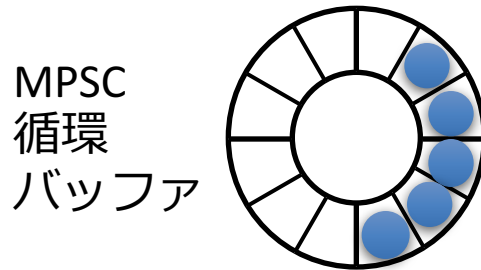
低オーバーヘッド



# Executor

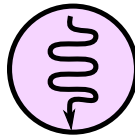
23

- キューから通信要求を取得 (Consumer)



4. キューから取出

**Executor**



5. 通信を指示

post() ↓

インターコネクト

普段はビジーウェイト

```
while (true) {  
    while (/*キューが空*/) {}  
    while (/*可視フラグがオフ*/) {}
```

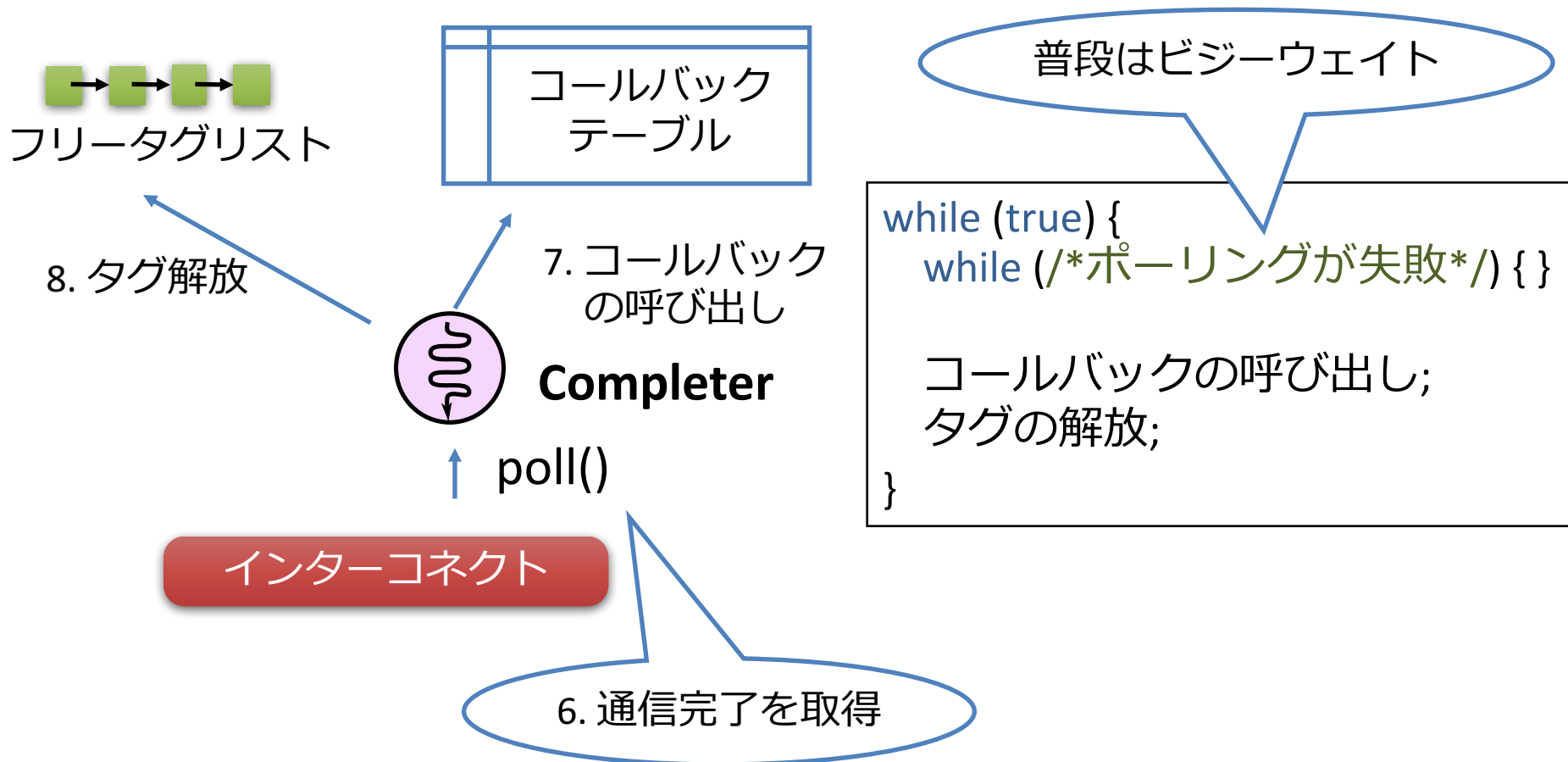
コマンドを実行;  
可視フラグをオフにする;  
headを進める;

```
}
```

# Completer

24

- ・ インターコネクトのポーリング関数を呼び出す





# 通信完了通知の手法

25

- 完了通知をコールバック関数で行う

- 既存処理系のよくあるAPI

```
gasnet_handle_t handle = gasnet_get_nb(/*...*/);  
gasnet_wait_syncnb(handle);
```

- 様々な通知手法
  - アトミック変数のセット
  - アトミック変数のRMW (read-modify-write)
  - 条件変数への通知 (OSレベル or ユーザーレベル)
- どれが最適かは利用者に依存
  - 全てAPIとしてカバーすることは難しい

# APIの特徴

26

- 「失敗可能」であること
- 「コールバック関数」によって通知
- RDMAアドレスのサイズ > ポインタのサイズ
- ローカルバッファのレジストレーションを強制
- 通信順序を保証しない
- 明示的なポーリングの排除

```
using process_id_t = /*integer*/;
struct remote_address { size_t offset; /*...*/ };
struct local_address { size_t offset; /*...*/ };
struct callback { void (*f)(void*); void* d; };
struct read_params {
    process_id_t src_proc;
    remote_address src_raddr;
    local_address dest_laddr;
    size_t size_in_bytes;
    callback on_complete;
};
bool try_read_async(const read_params&);
```

# 各インターコネクトごとの実装詳細

27

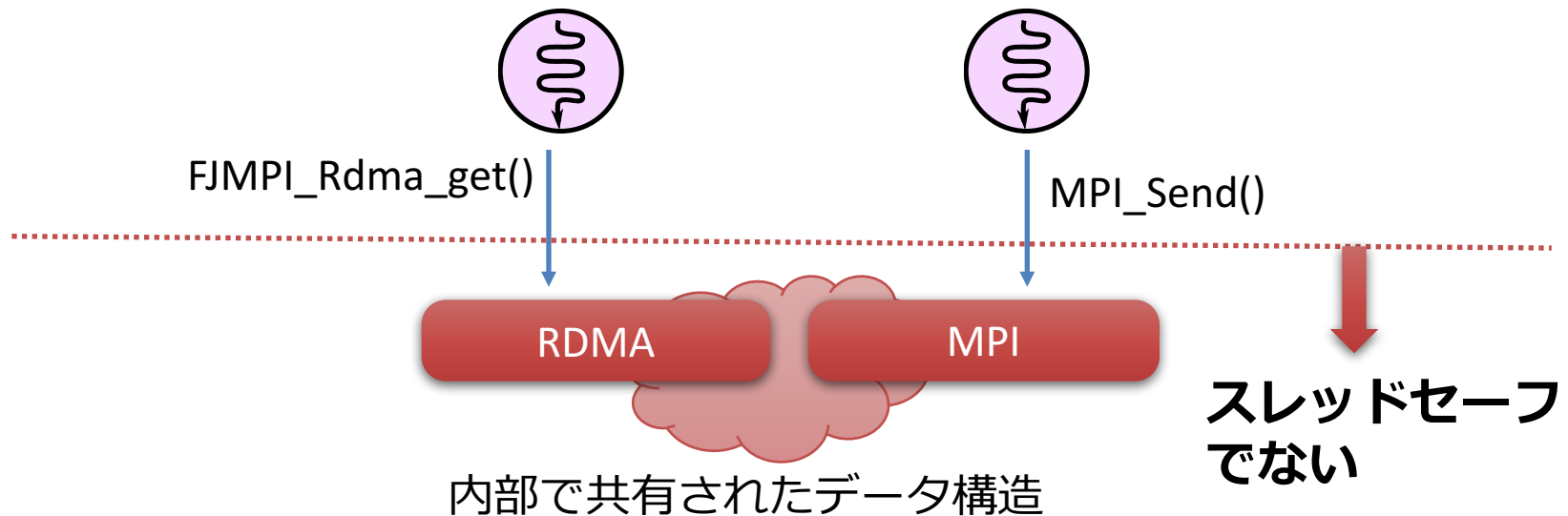
- 現在対応しているインターコネクトAPI
  - Tofu
  - InfiniBand Verbs
- 互換性のための実装
  - MPI-1
  - MPI-3

# TofuのRDMA API

28

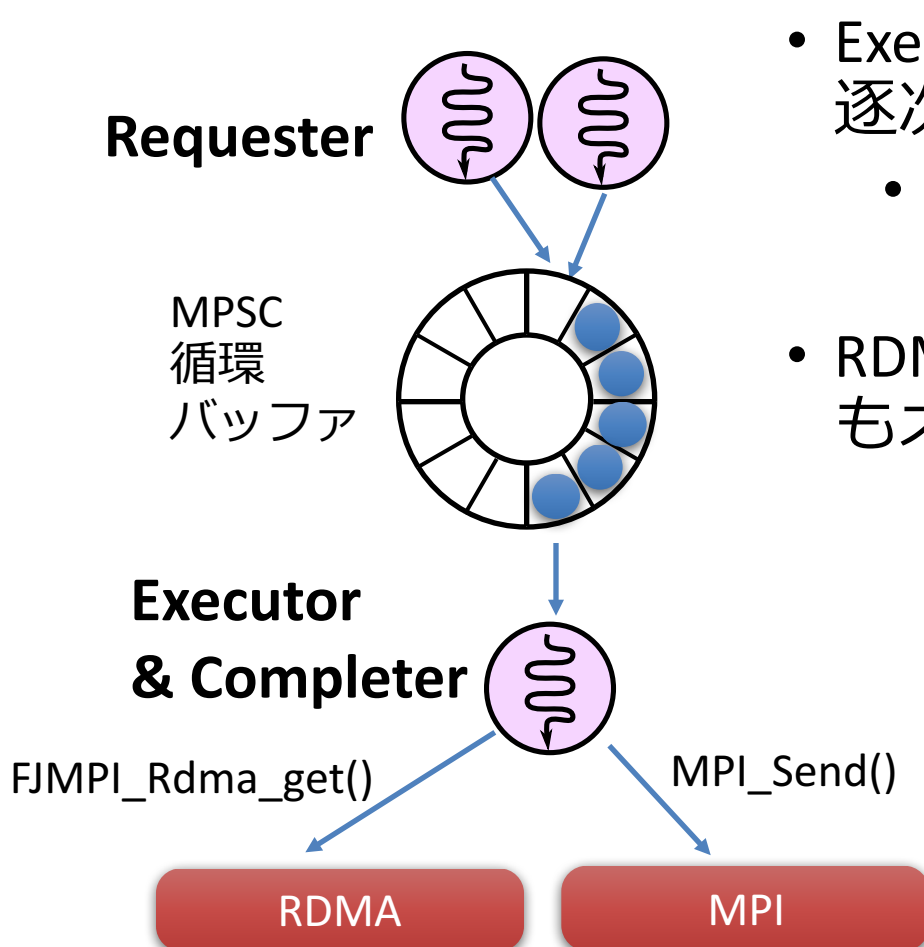
- スレッドセーフでない

- 扱うNICが違っていても同時に操作できない  
= NICが複数あるが、API自体には並列性がない
- 通信要求とポーリングも並列実行不可
- RDMAとMPIでデータ構造を共有している



# Tofu用実装

29

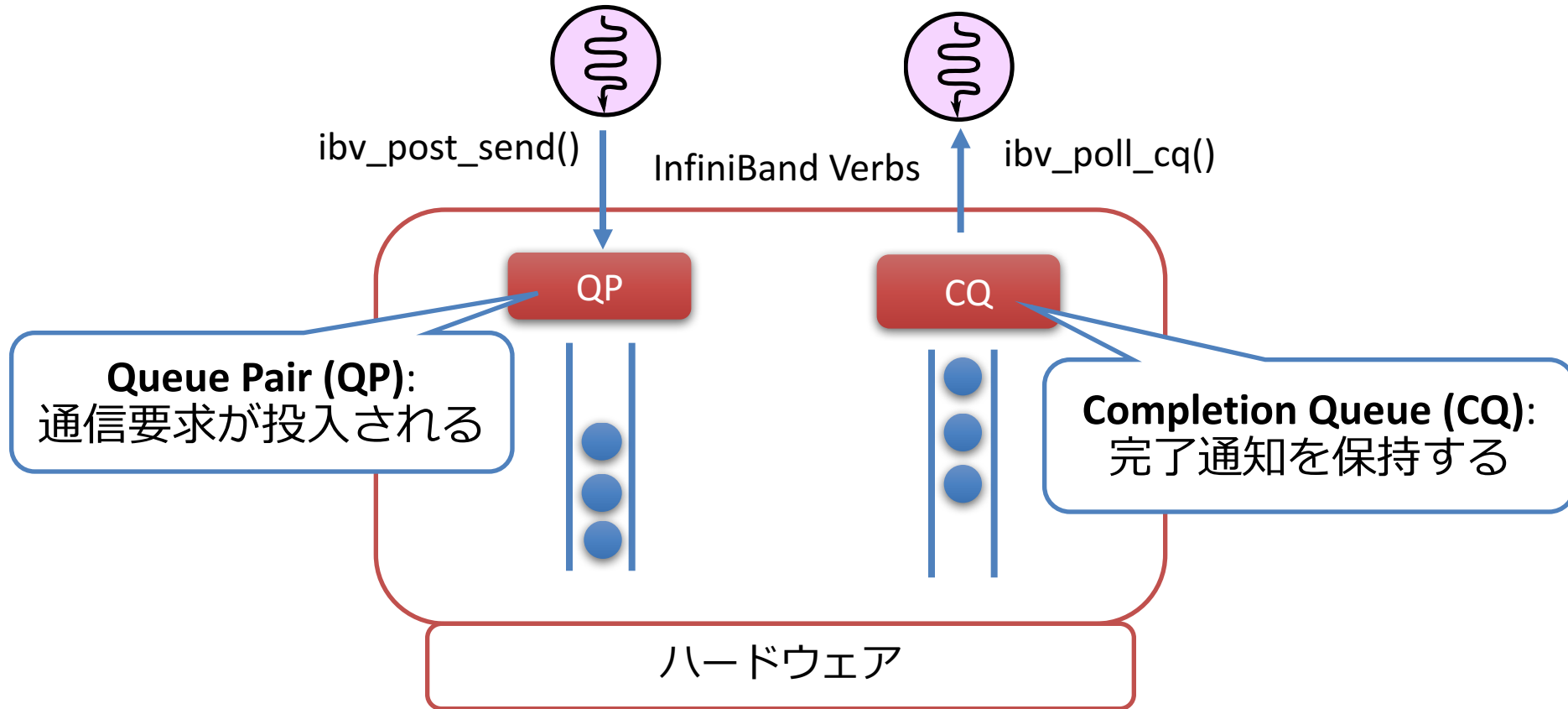


- ExecutorとCompleterの両方を逐次化
  - 単一の通信専用スレッドを用意
- RDMAだけでなくMPI呼び出しもオフロード

# InfiniBand Verbs (IBV)

30

- 全てのAPIが**スレッドセーフ**を保証



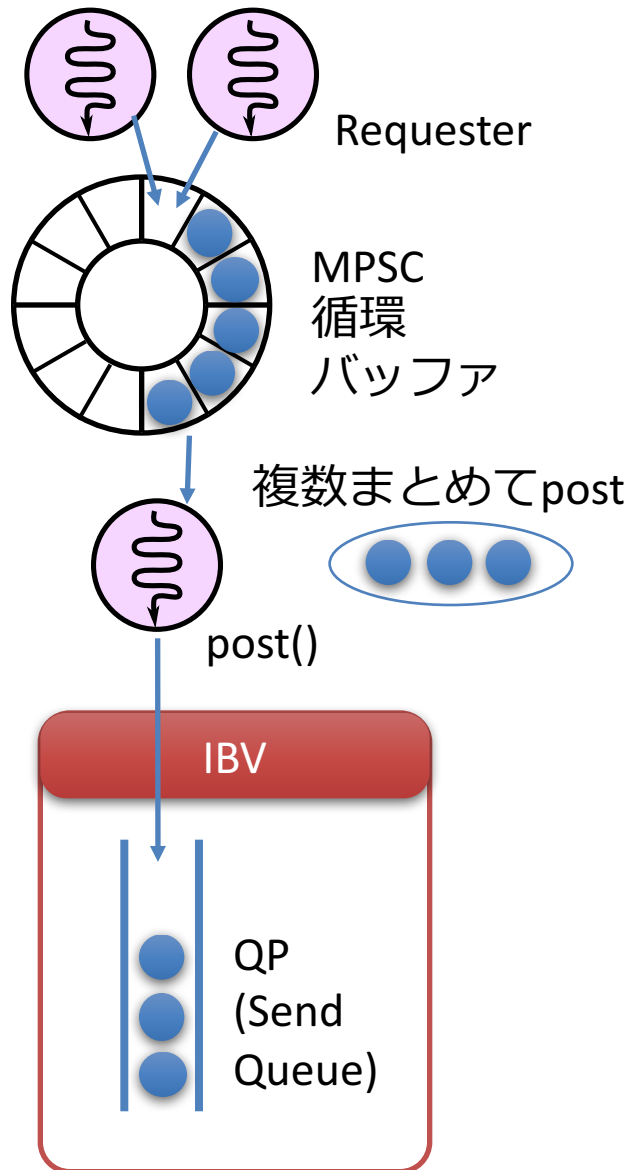
# post()のソフトウェアオーバーヘッド

31

- `ibv_post_send`の流れ
  - 1. **スピンロック**を獲得
  - 2. アドレス情報を書き込む
  - 3. ハードウェアに通知（doorbellを鳴らす）
  - 4. **スピンロック**を解放
- `ibv_post_send`自体に最短で数百クロック
  - 仮に500クロックだとして, 3GHzのCPUで呼び出せる回数は $6 \times 10^6$ 回/秒
  - InfiniBandの公称メッセージレートは $\sim 100 \times 10^6$ /秒
    - 1回の`ibv_post_send`の呼び出しで複数投入する必要
  - 1メッセージあたり1回の関数コールではオーバーヘッドが大きすぎる

# IBVにおけるメッセージ集約

32



- 複数の通信要求を一度にpostする方法
  - ユーザに集約してもらう
    - 一般にはいつ通信が発生するか分からないので困難
  - **システムが自動的に集約**
    - キューにためることで可能



# 評価手法

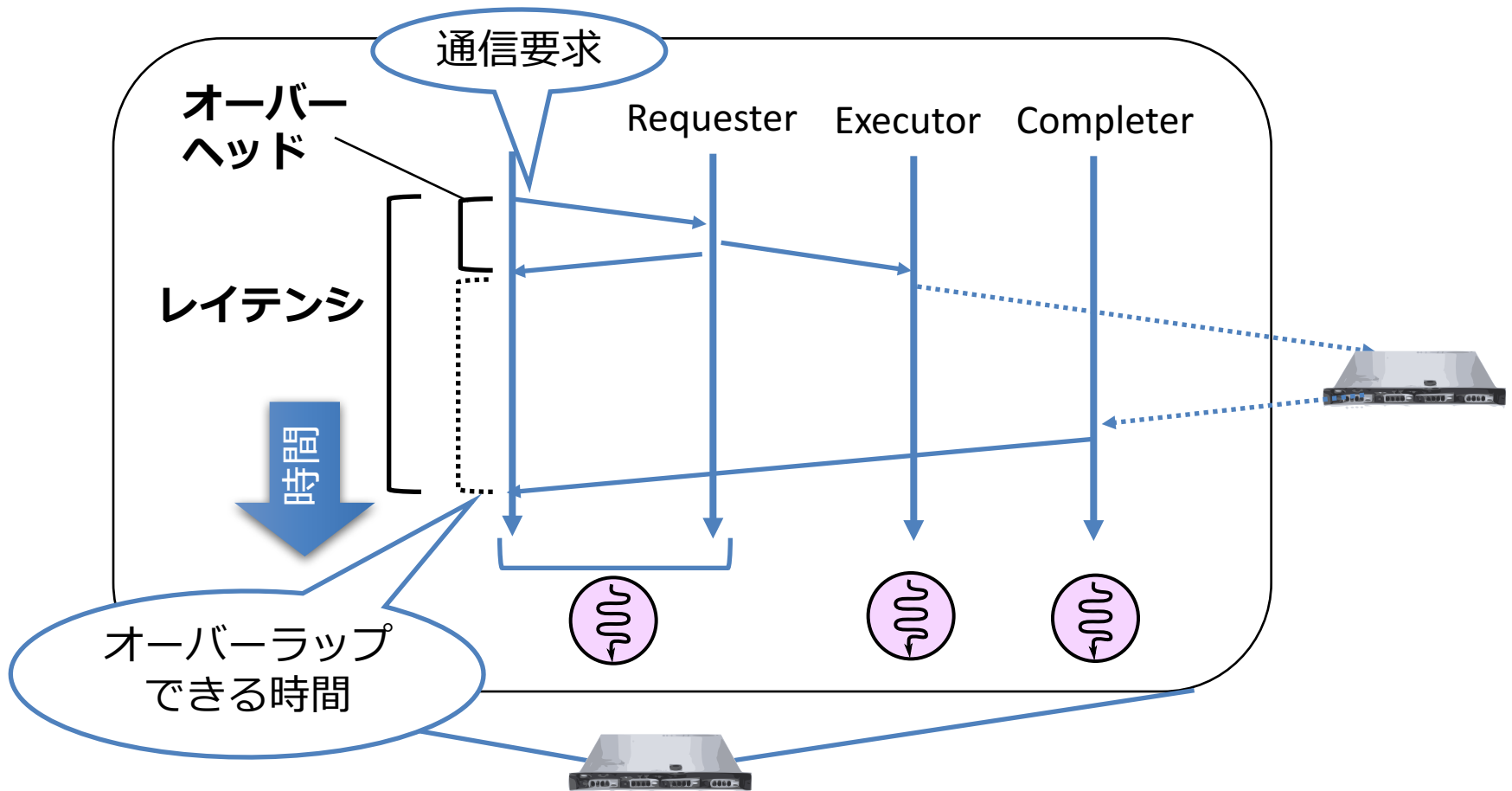
33

- マイクロベンチマークによる評価
  - 2ノード間でRDMA READを繰り返す
  - 1ノードあたり1プロセス
  - スレッド数は可変
- Tofuでの実験
  - 東大情報基盤センターのFX10
    - CPU: SPARC64 IXfx, 16コア/ノード
  - ExecutorとCompleterが共有する1スレッド
- InfiniBandの実験環境
  - 東大情報基盤センターのKNSCクラスタ
    - CPU: Intel® Xeon® E5-2680 v2, 2ソケット×10コア/ノード
    - InfiniBand FDR 2-port (ただし1ポートのみ使用)
  - Executor, Completerにそれぞれ1スレッド

# レイテンシとオーバーヘッドの測定

34

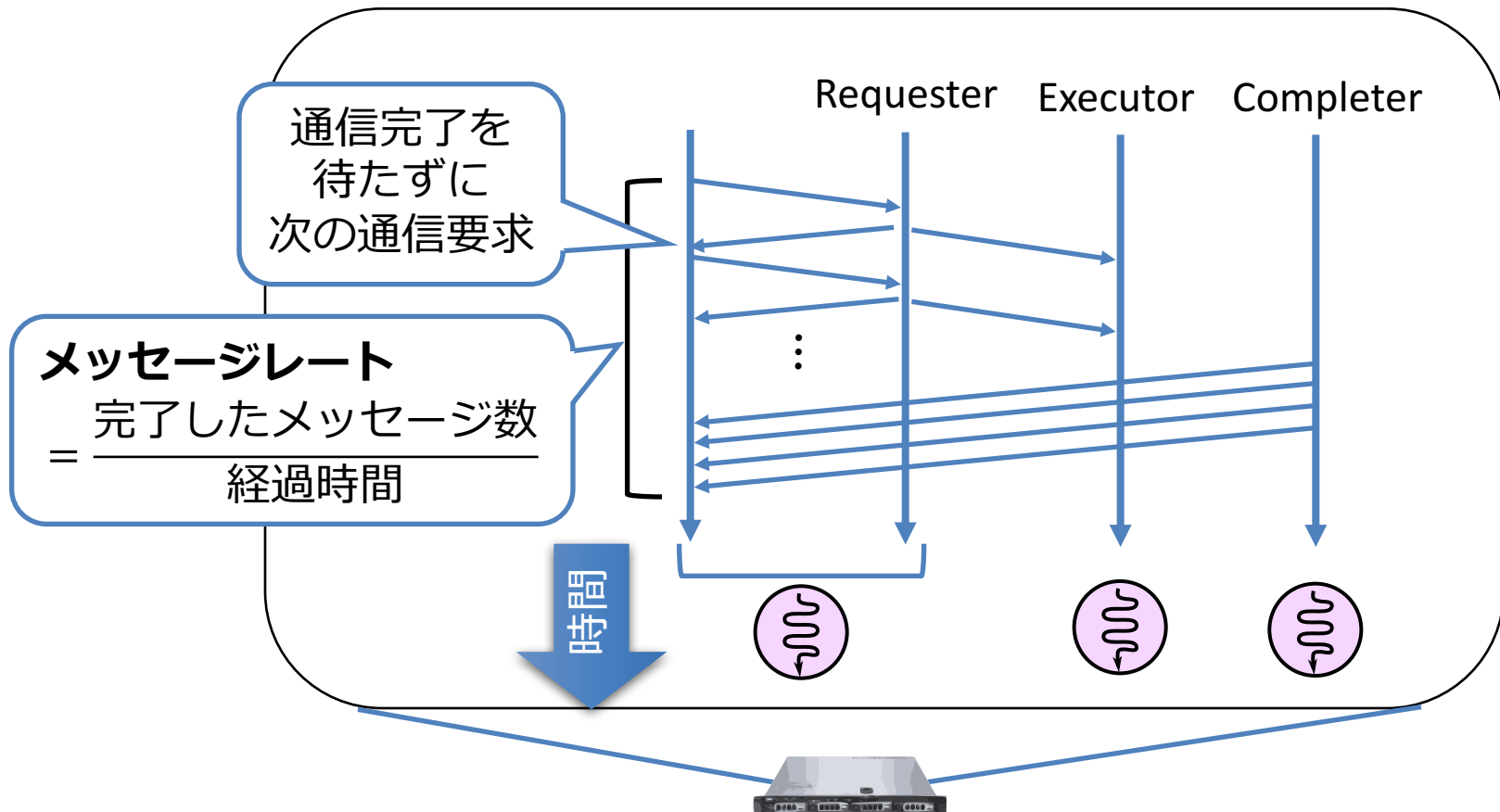
- ベンチマークの流れ
  - 各スレッドがRDMA READを要求
  - 完了したらまたRDMA READを要求して繰り返す



# メッセージレートの測定

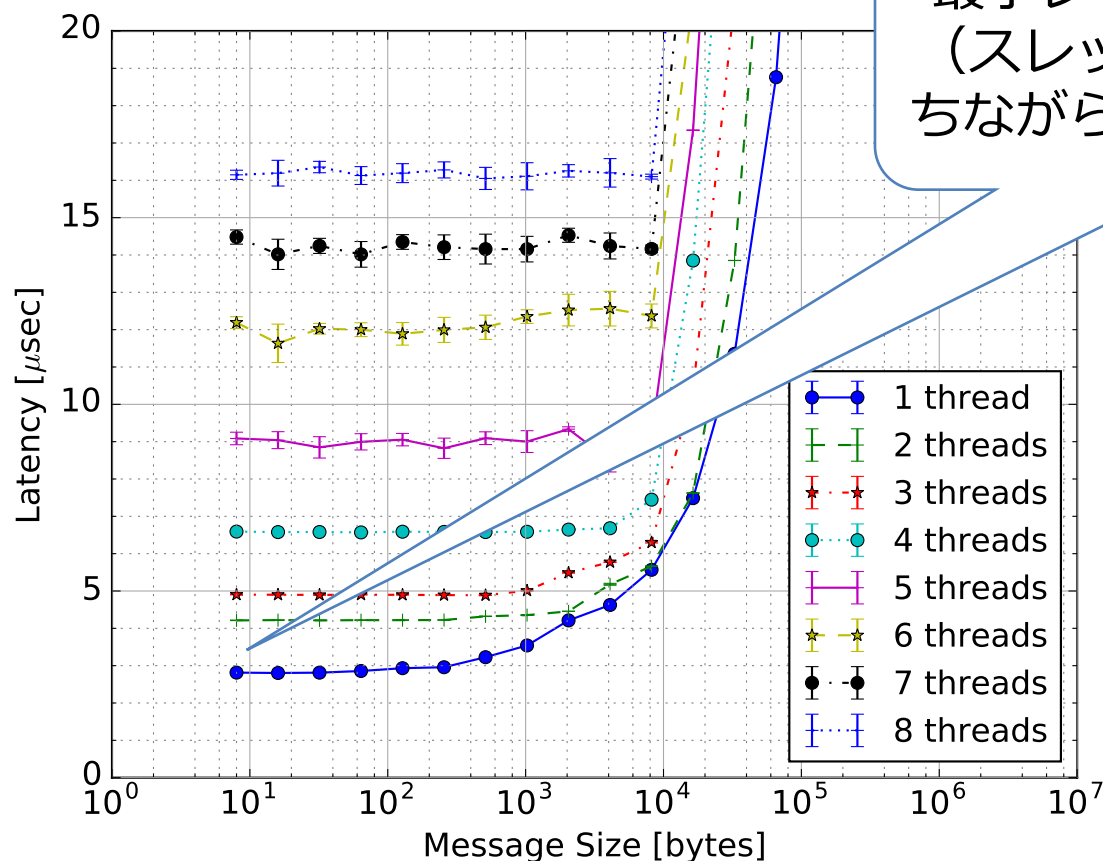
35

- ベンチマークの流れ
  - 各スレッドがRDMA READを要求
  - 完了を待たずに次のRDMA READを要求



# Tofuにおけるレイテンシ

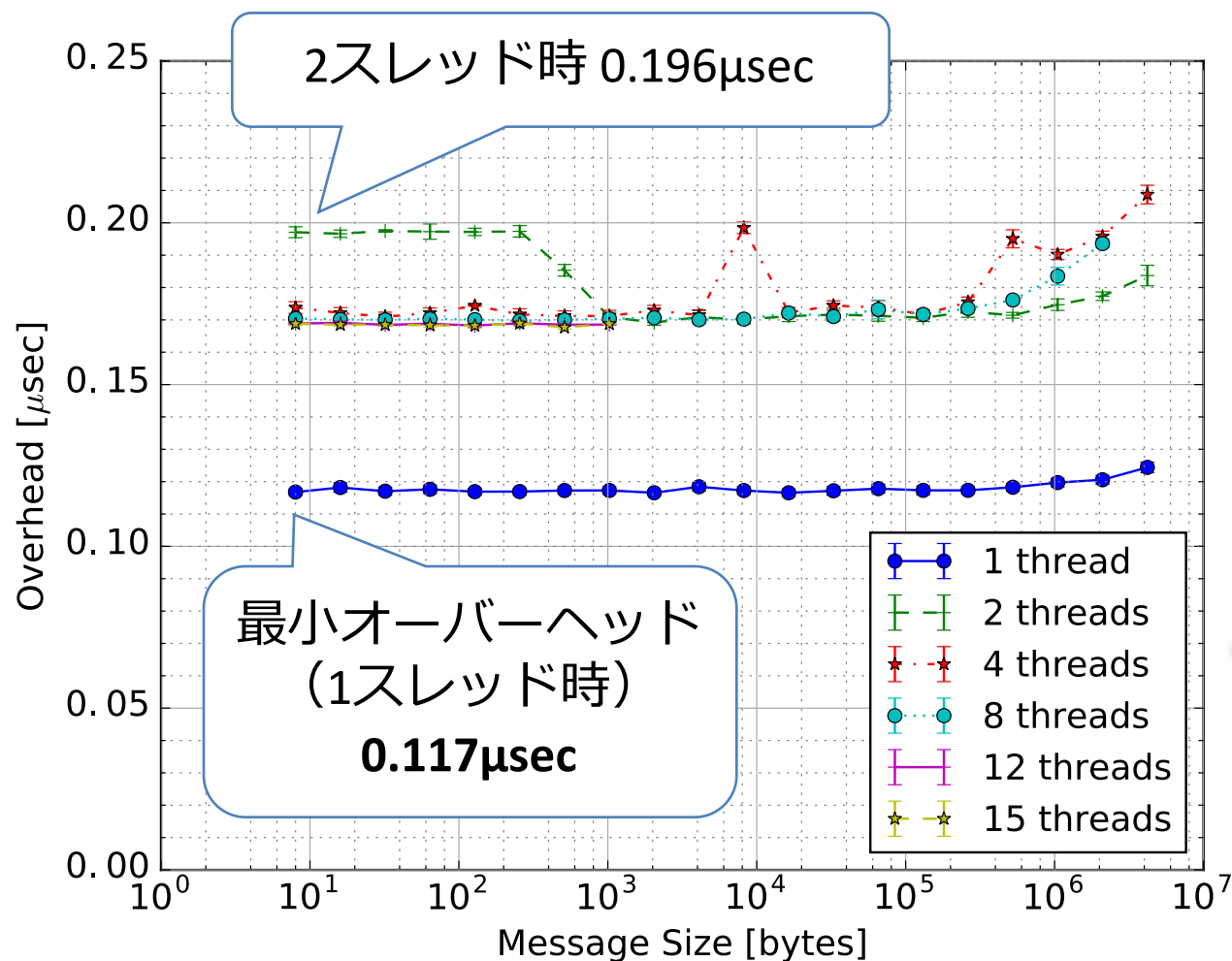
36



参考: ACP基本層 [野瀬 et al. '15] でのレイテンシ増大は **+88%**  
(ただしアドレス変換のオーバーヘッドを含む)

# Tofuにおけるオーバーヘッド

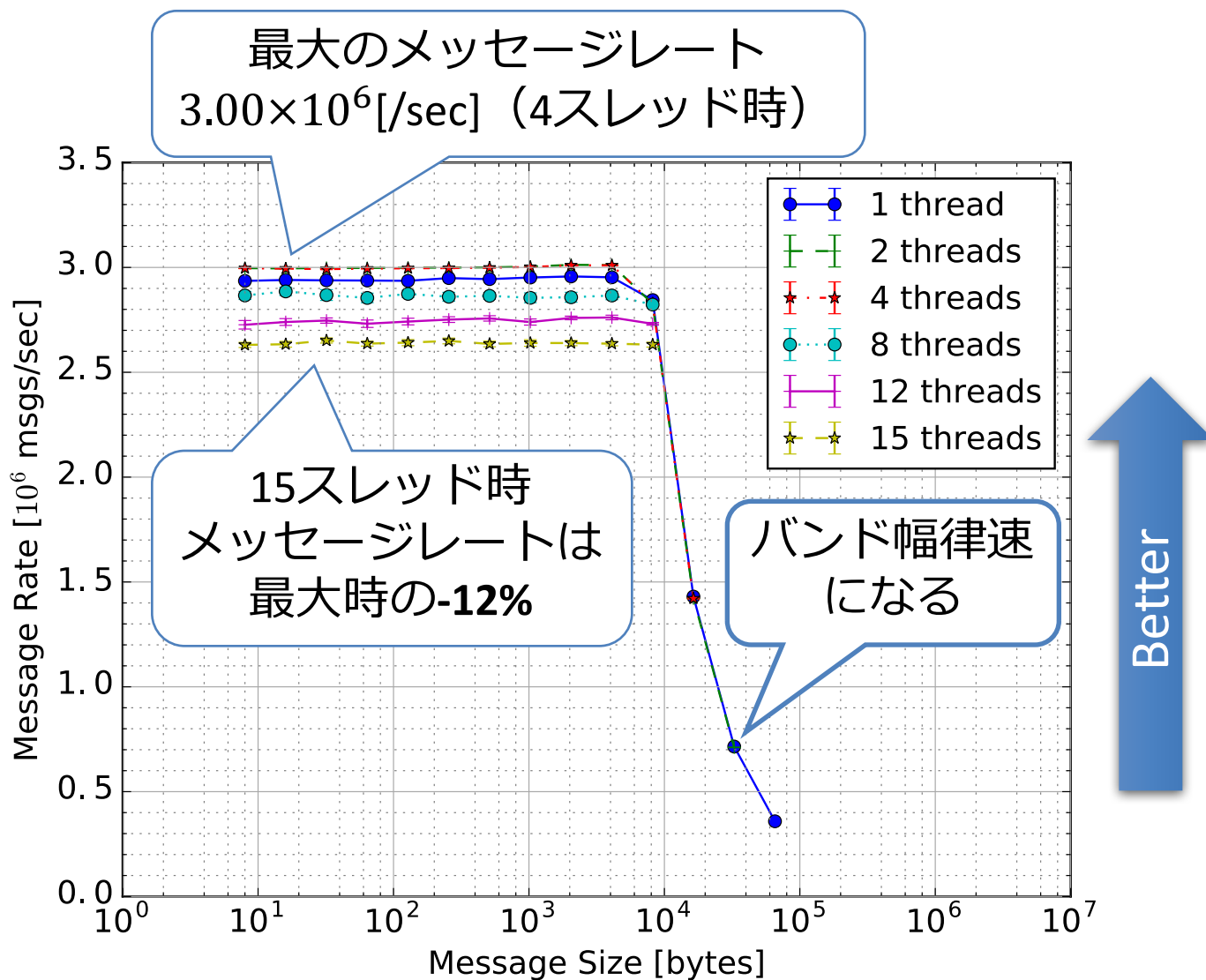
37



Better

# Tofuにおけるメッセージレート

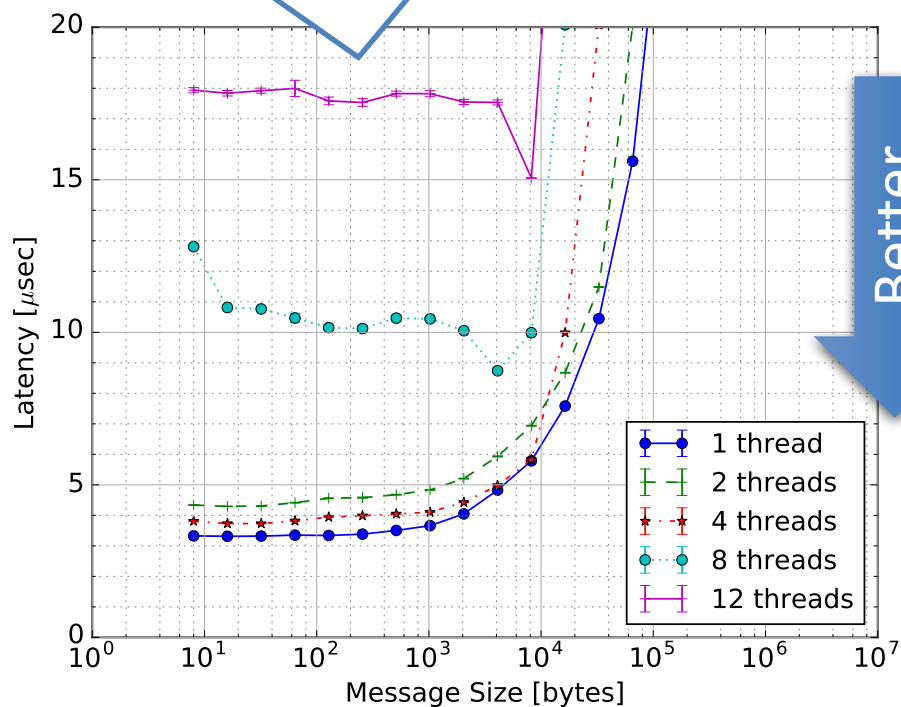
38



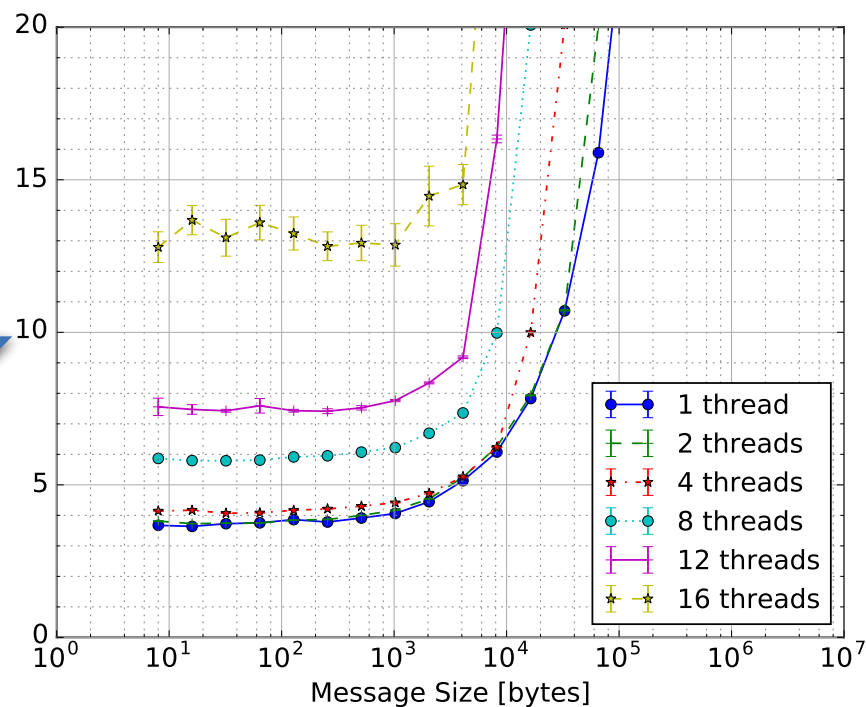
# IBVにおけるレイテンシ

39

スレッド数増加時、スピンロック  
が原因で急激にレイテンシ増大



Better



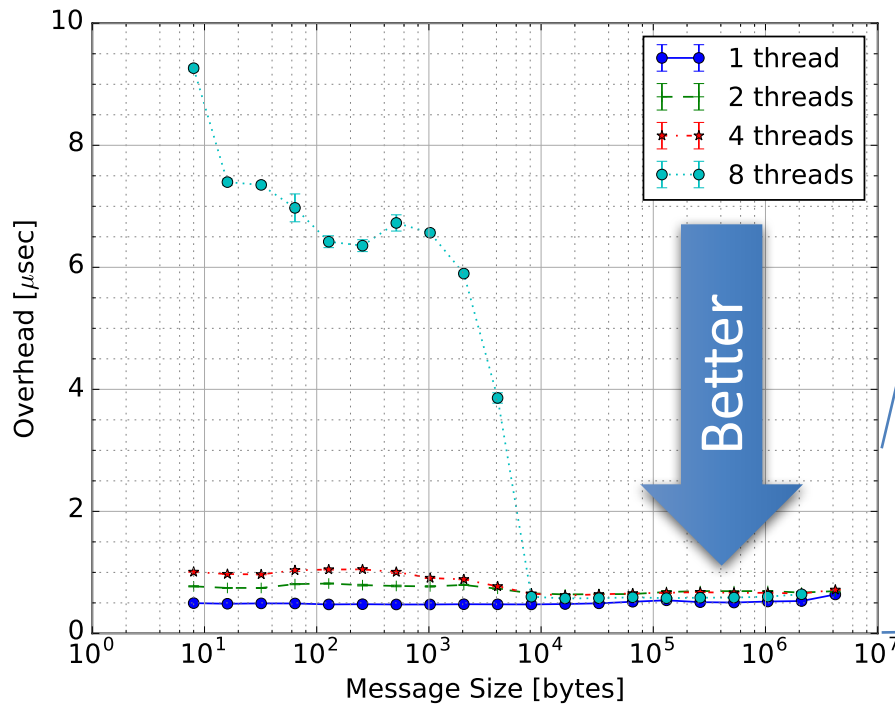
(a) オフローディングなし

(b) オフローディングあり

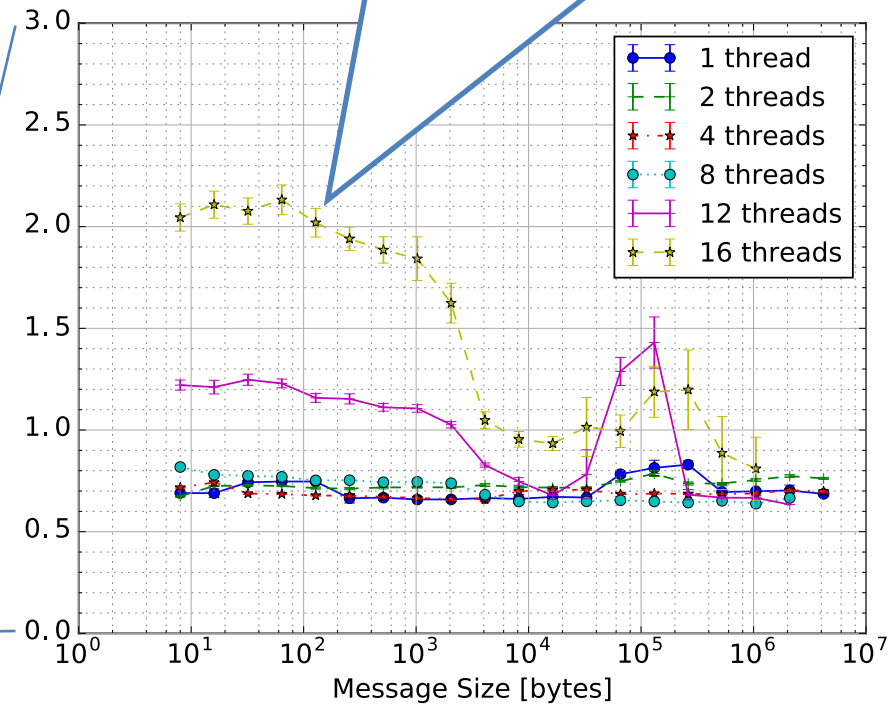
# IBVにおけるオーバーヘッド

40

Tofu用実装と比べて増え幅が大きい  
(タグ管理時のスピンロックによる)



(a) オフローディングなし

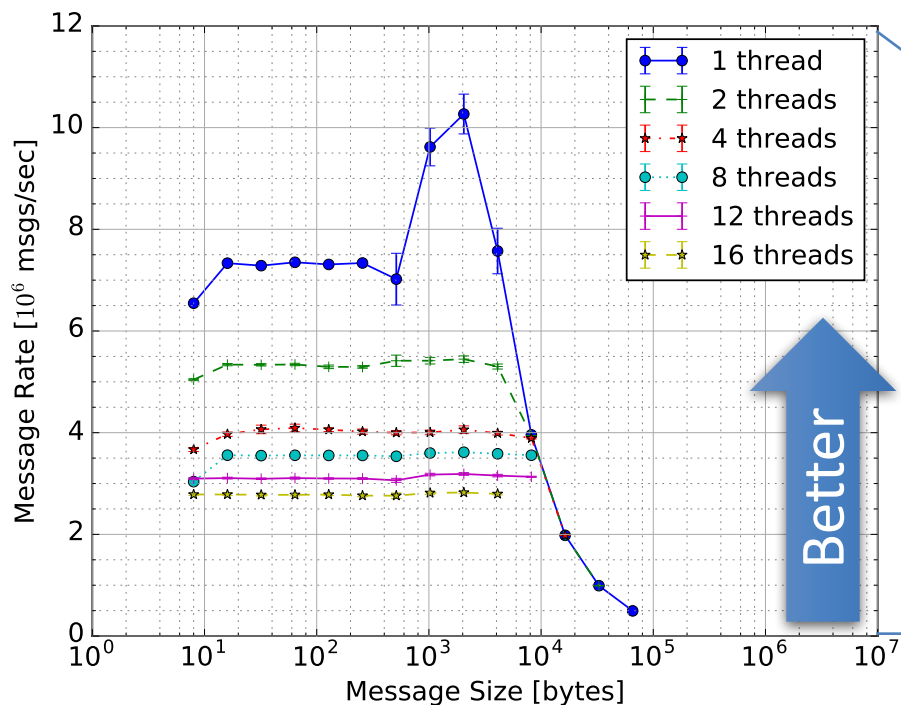


(b) オフローディングあり



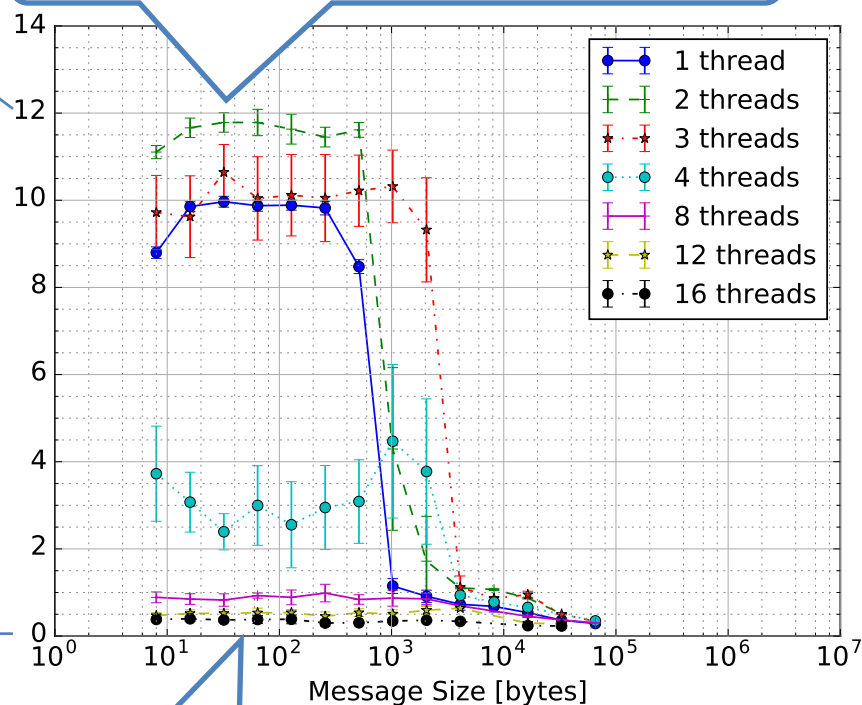
# InfiniBandにおけるメッセージレート

41



(a) オフローディングなし

安定して約 $10 \times 10^6$  [/sec] 出ている



(b) オフローディングあり

マルチスレッド性能には問題あり  
(タグ管理時のスピンロックによる)

# まとめ

42

- 低水準通信レイヤーAPIの定義
  - ハードウェアの仕様に近づけた設計
- 低水準通信レイヤーでのオフローディング
  - MPSC循環バッファ
  - 多数スレッド時のメッセージレート向上
  - 実装の工夫でオーバーヘッド低減
- オフローディングを利用した通信集約
  - InfiniBandにおいてメッセージレート向上

# 今後の方針

43

- IBVでのマルチスレッド性能の改善
  - SPMC循環バッファによってタグ管理
- 低負荷時のオフロードを回避
  - レイテンシ削減のため
- Executor, Completerのスピンウェイトを回避
  - 条件変数の併用
- 実アプリケーションでの性能評価