HPC向け通信隠蔽技術の効果の検証

九州大学情報基盤研究開発センター
南里　豪志

第24回AT研究会オープンアカデミックセッション（ATOS24）
2020年 8月 19日（水）

自己紹介

- 氏名：南里豪志
- 所属：九州大学 情報基盤研究開発センター
- 研究分野：
  高性能計算の基盤技術
  - 主に並列計算における通信効率化技術

- 仕事：
  スーパーコンピュータの運用
- 趣味：
  ジョギングを少々
  - 昨年、2度目の挑戦でようやくフルマラソン完走

現在までの主な成果

- 集団通信のアルゴリズム自動選択技術
  - プロセスが配置されたノードの位置等、実行時の状況に応じた選択
- 省メモリ通信ライブラリ
  ACP (Advanced Communication Primitives)
  - https://github.com/Post-Peta-Crest/ACP
  - CRESTプロジェクト（2011-2016、九大、富士通、九州ISIT）で開発
  - 暗黙的な通信バッファ不要
  - PGAS (Partitioned Global Address Space) モデルによる
    簡易な片側通信インタフェース
- 連成計算支援フレームワーク
  CoToCoA (Code To Code Adaptor)
  - https://github.com/tnanri/cotocoa
  - 科研（萌芽）（2017-2019、代表 加藤（東北大））で開発
  - 2つのMPIプログラムを結合した連成計算を支援
  - 元のプログラムへの書き換えを最小限にする結合インタフェースを提供

今回の話

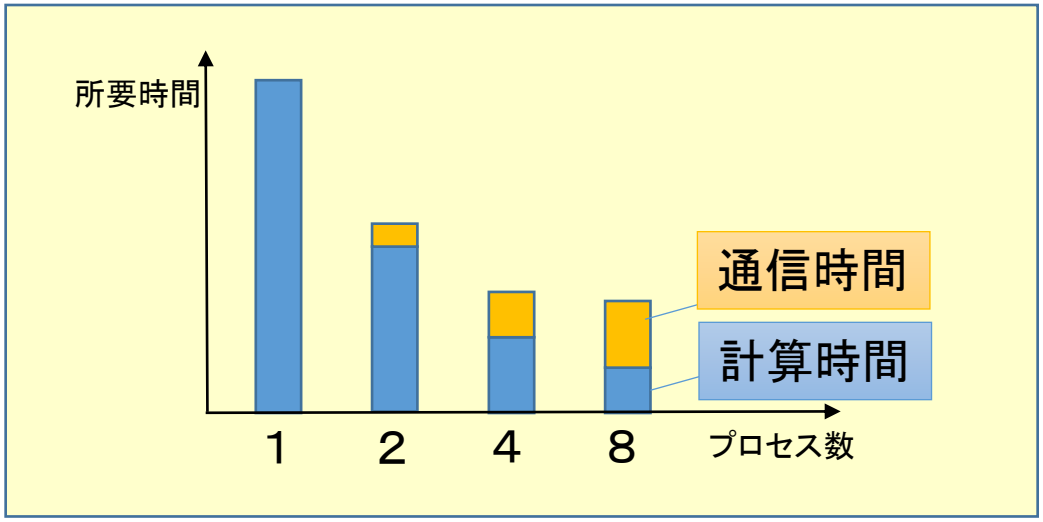
- 高性能計算向け通信隠蔽技術の効果の検証
  - どのような条件で効果が得られるかを調べ、
    今後の通信最適化技術の研究開発に活かしたい

- トピック
  - 非ブロッキング一対一通信 vs 片側通信
  - オフロード型非ブロッキング集団通信

- 少し言い訳
  - 今までに何度か話した内容なので、どこかで聞いたことあるなぁ、
    という方も、いらっしゃると思います。
  - 一応、2020年 8月時点で、九州大学のスーパーコンピュータ ITOで使える
    （ほぼ）最新の通信ライブラリで計測しなおした結果をお見せするので、
    ご容赦ください。

背景：通信時間を隠したい

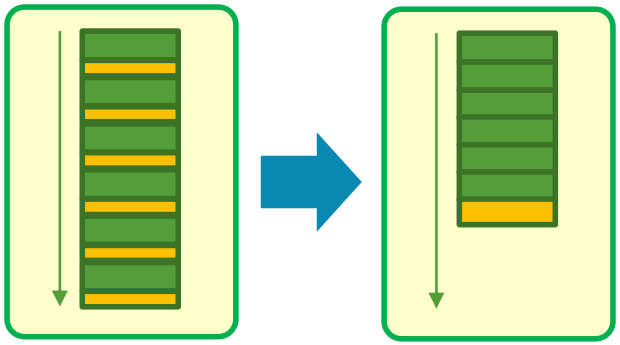
- 高並列計算の内訳
  - 計算時間： プロセス数に応じて減少（並列化できた部分）
  - 通信時間： プロセス数が増えても変わらない、もしくは増加



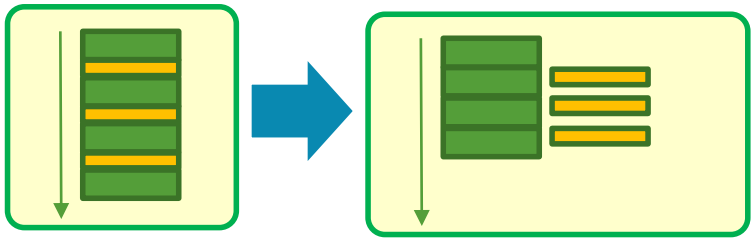
プロセス数に応じて通信時間の比率が増大

通信時間を削減する方法

- 通信回避
  (communication avoiding)
  - 通信の回数と量を減らす
    - 例） Communication-Avoiding CG Method
  - ほぼアプリケーション依存



- 通信隠蔽
  (communication hiding)
  - 通信と計算を同時進行して、見かけ上の通信時間を隠す
  - アプリケーションにも依存するが、**<u>通信基盤からの支援</u>**も重要



通信最適化技術の研究者として興味

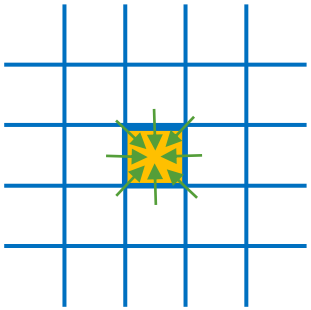
あらためて、今回の話

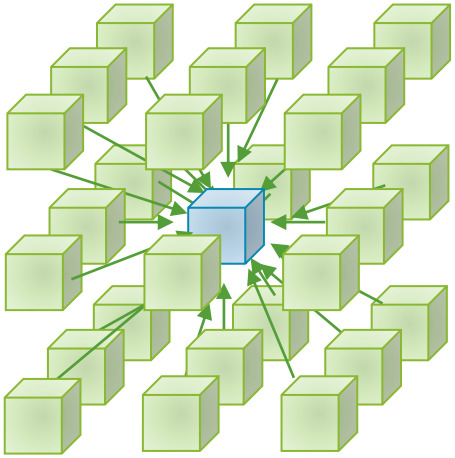
- 一対一通信の隠蔽技術の効果
  - Stencil計算におけるHalo通信を題材に、通信隠蔽の効果を評価
    - 領域を分割した並列計算における袖領域の交換
  - 非ブロッキング一対一通信 vs 片側通信

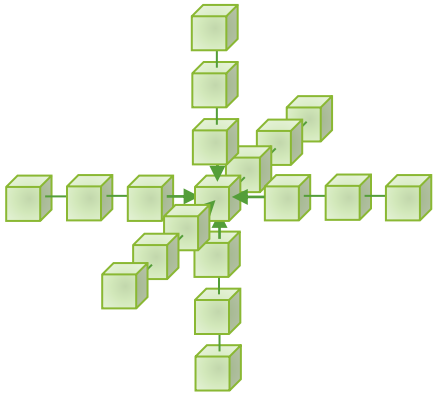
- 非ブロッキング集団通信のオフロード技術による隠蔽効果
  - インターコネクトネットワークのスイッチに集団通信を依頼
  - 自作ベンチマークによる検証

Stencil計算

- 配列の各要素を、周囲の要素の値に基づいて更新
  - 例）



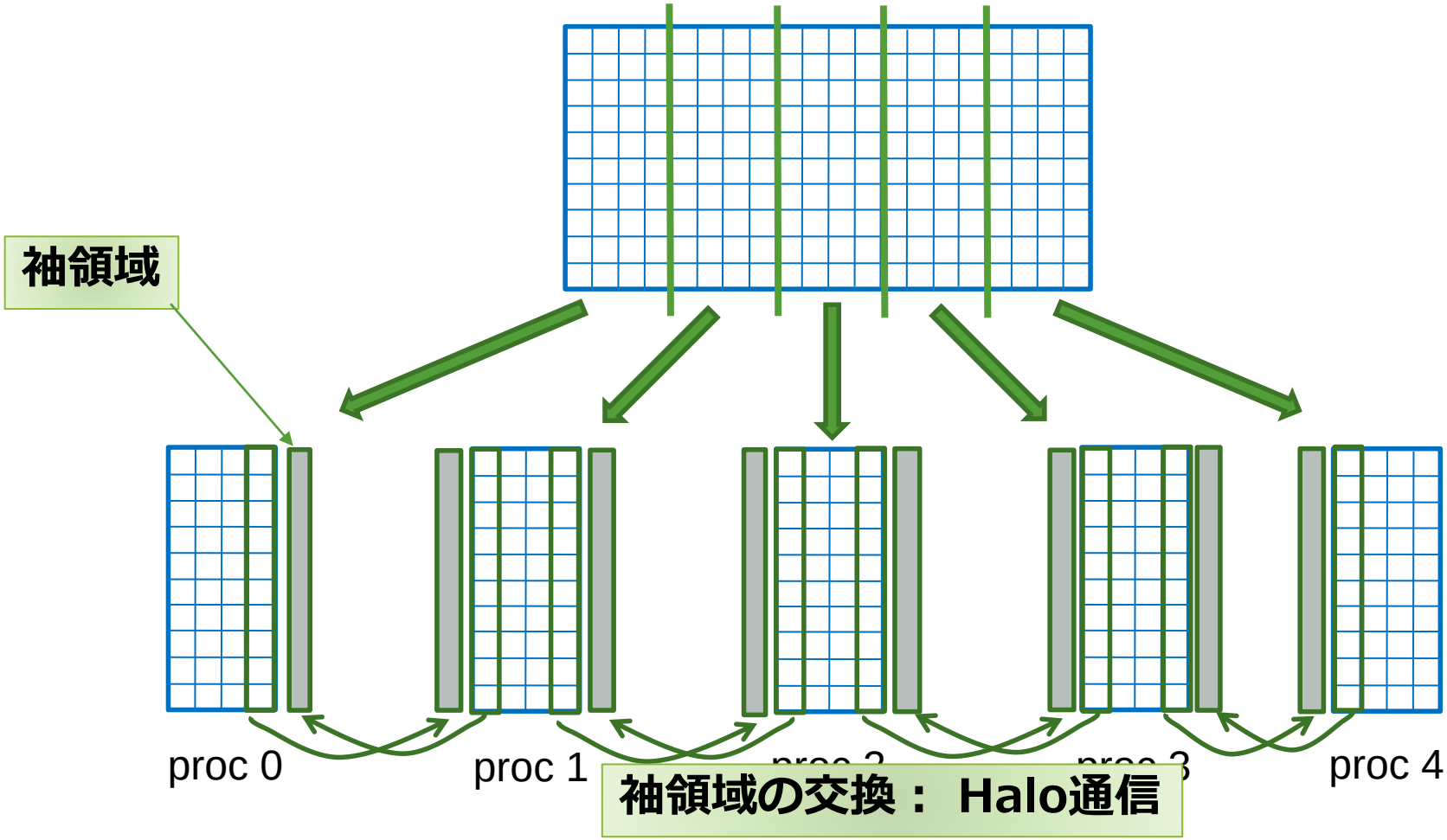




- プロセス並列化
  - 1次元～n次元のプロセス空間に分割
  - 隣接プロセス間で、境界部分（袖領域）の交換のために通信

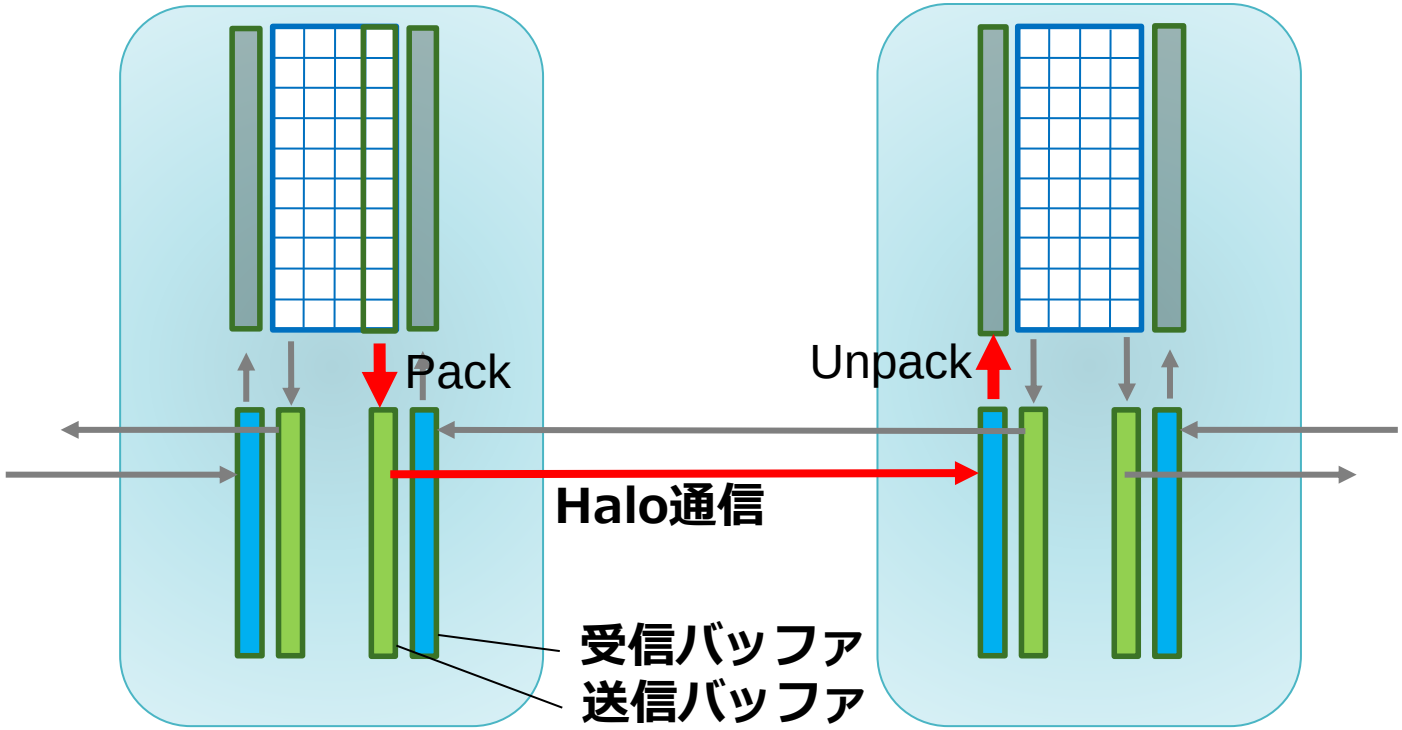
今回の例：
2次元 Stencil計算を 1次元分割

- 境界に1次元の袖領域



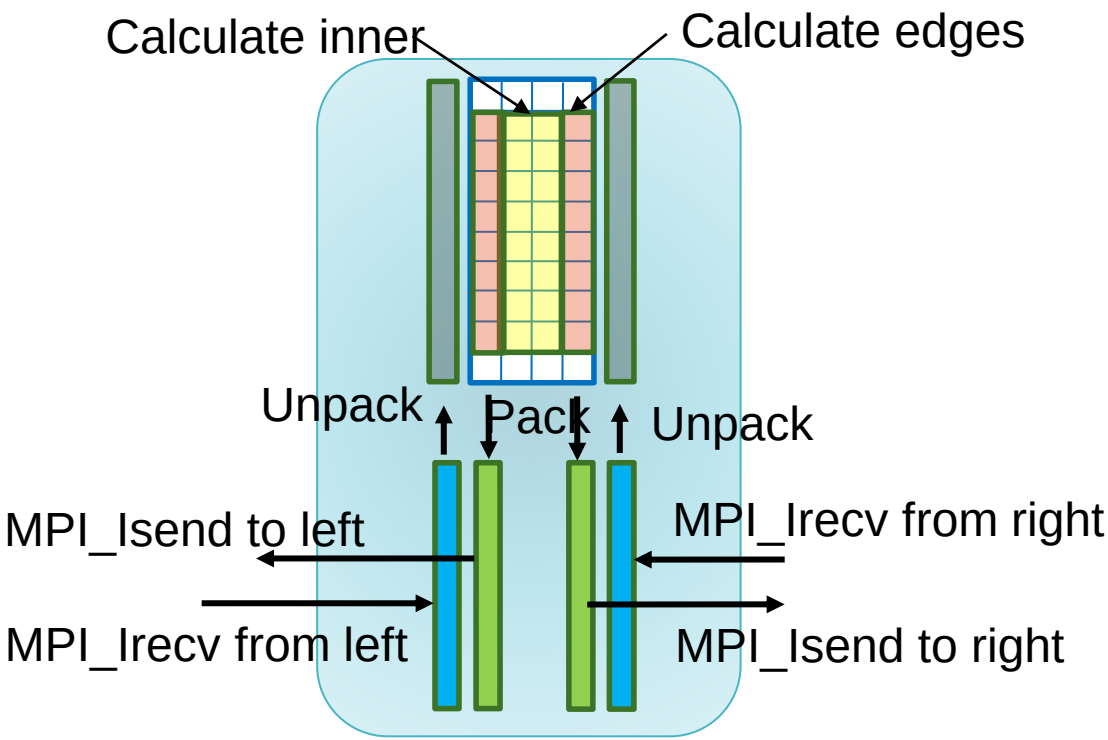
Halo通信における Pack / Unpack

- 不連続なアドレスに配置された要素群による袖領域の通信
  - Pack: 袖領域の要素を送信用領域にコピー
  - Unpack: 受信した要素を袖領域にコピー
  - 少し言い訳
    - 本当は Stencil計算の1次元分割では、分割方向を変えれば不要
    - 実際の問題ではほぼ必要になるので、今回は Pack / Unpack付きで実験



MPI_Isend/Irecv による Halo通信

```
Setup arrays

MPI_Irecv from left
MPI_Irecv from right

for (steps)
  Pack
  MPI_Isend to left
  MPI_Isend to right

  Calculate inner

  MPI_Waitall
  Unpack
  MPI_Irecv from left
  MPI_Irecv from right

  Calculate edges
```



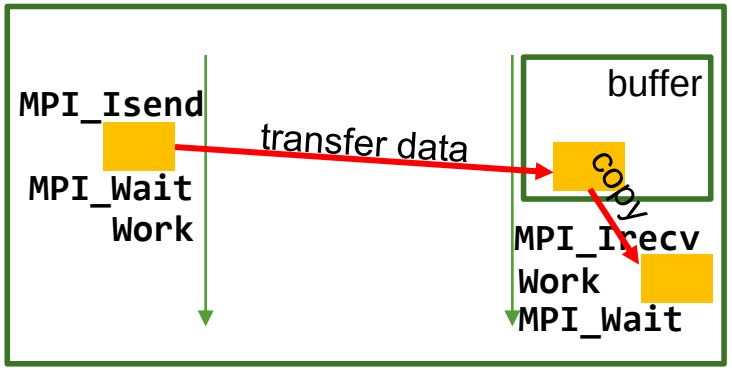
MPI_Isend / Irecv の中身

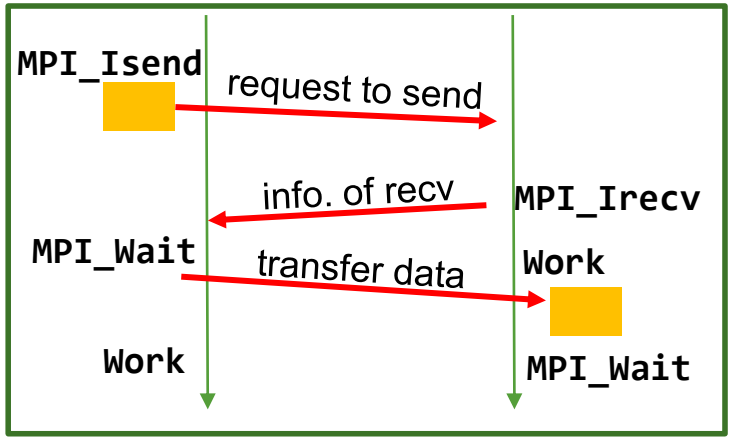
- 基本的に、二種類のプロトコルを利用

- Eagerプロトコル
  - 受信プロセスを待たずに送信を完了可能
    ⇒ **高い通信隠蔽効果**
  - **通信サイズが受信バッファの大きさで制限**

- Rendezvousプロトコル
  - **大きなメッセージも送信可能**
  - 受信プロセスの受信命令を待って送信開始
    ⇒ **低い通信隠蔽効果**

- 通常、数KBを閾値にして
  使用プロトコルを切り替え

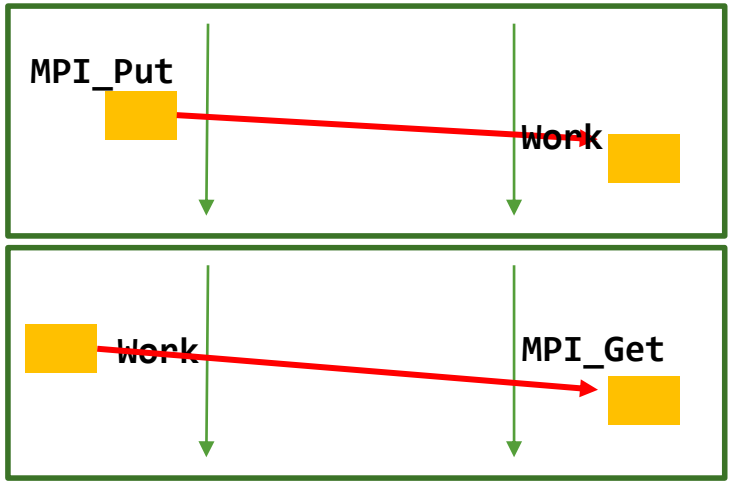




メッセージサイズが大きい場合、通信隠蔽効果が得られにくい

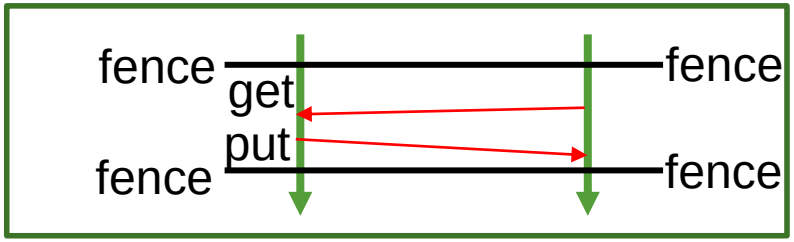
片側通信

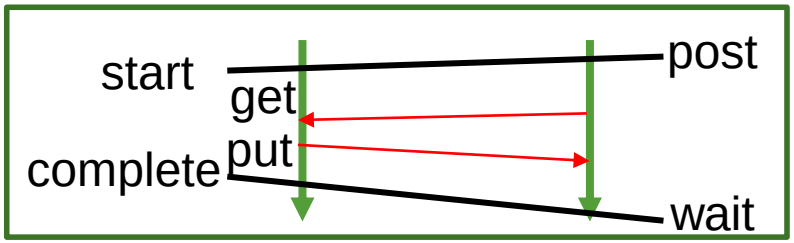
- 相手のプロセスの状況によらず、データ転送
  - MPI_Put: 書き込み
  - MPI_Get: 読み出し
- メッセージサイズに関係なく通信隠蔽可能
- 通常、同期用の通信が別途必要
  - MPI_Put
    - 上書きを許可
    - 書き込み完了を通知
  - MPI_Get
    - データの生成を通知
    - 読み出し完了を通知



片側通信の同期（１）Active Target

- 転送元プロセスと転送先プロセスの間で同期しながら対象領域を管理
  - MPI_Win_fence
    - 全プロセスの対象領域への読み書きの許可 / 禁止
  - MPI_Win_post / start / complete / wait
    - 読み書きされる側：
      - post：対象領域の読み書き許可
      - wait：読み書き完了を待って対象領域の読み書き禁止
    - 読み書きする側：
      - start：読み書き開始
      - complete：読み書き終了

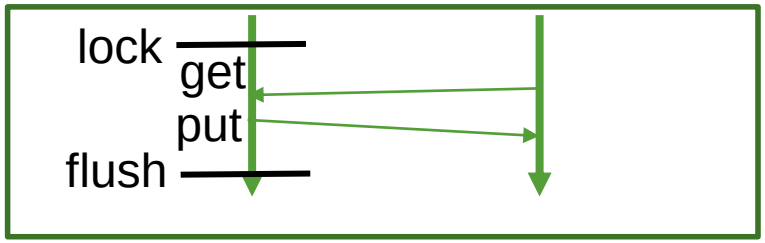




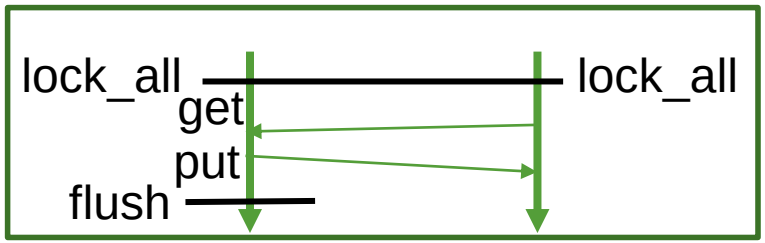
プログラムは比較的書きやすいが、必要以上に同期が発生する可能性あり

片側通信の同期（2）Passive Target

- 転送元プロセスと転送先プロセスは非同期に領域を管理
  - MPI_Win_lock / unlock / flush
    - lock：領域への読み書き権を取得
    - unlock：領域への読み書き権を返上
    - flush：発行した読み書きの完了待ち



  - MPI_Win_lock_all
    - 全プロセスに、領域への読み書きを許可
      - 誰からどのタイミングで読み書きされるか分からない
        ⇒ 共有メモリに近い状態



プログラマが頑張れば同期を最小限に出来るが、プログラムが分かりにくくなりやすい

Active TargetによるHalo通信

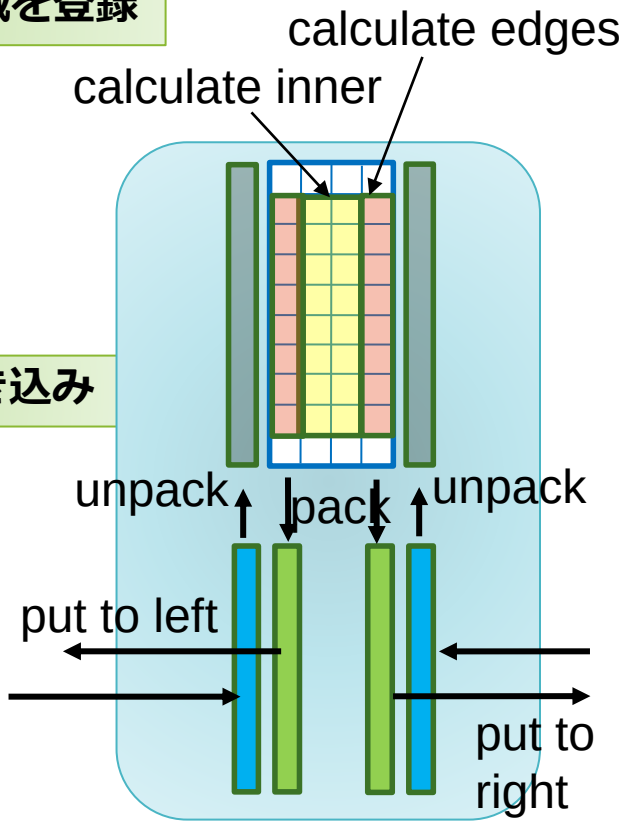
```
setup arrays
MPI_Win_allocate(buf, win)

for (steps)
  pack
  MPI_Win_fence(win)
  MPI_Put DATA to left & right

  calculate inner

  MPI_Win_fence(win)
  unpack

  calculate edges
```

他プロセスに読み書きを許す領域を登録

領域への読み書きを同期

左右のプロセスにデータ書き込み

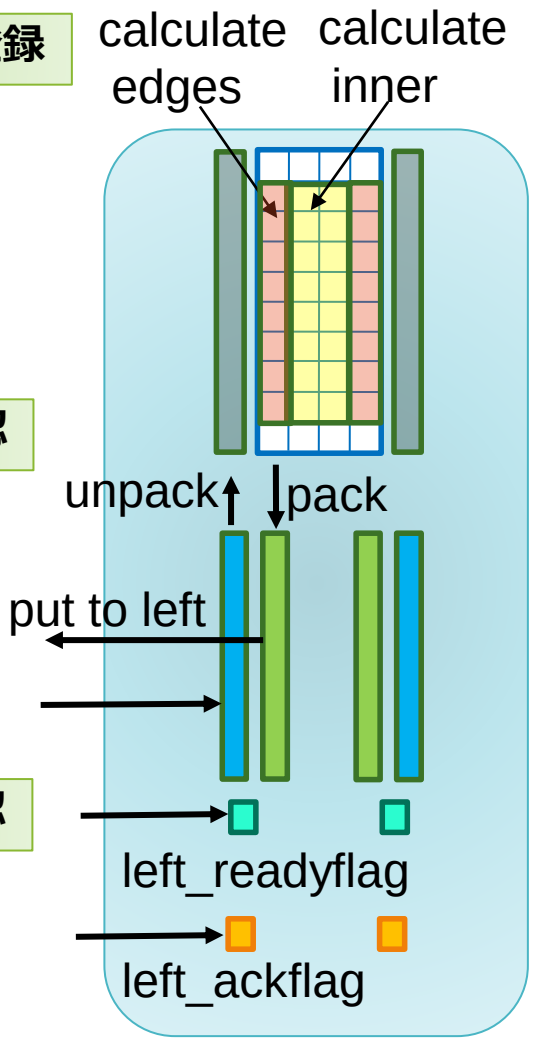
領域への読み書きを同期



Passive TargetによるHalo通信

```
setup arrays
MPI_Win_allocate(buf, win)
MPI_Win_lock_all(win)
setup flags

for (steps)
  pack
  MPI_Win_flush_all(win)
  while (...)
    if (left_readyflag == 1)
      MPI_Put DATA to left/right
      MPI_Put ACK to left

  calculate inner elements

  while (...)
    if (left_ackflag == 1)
      unpack DATA
      MPI_Put READY to left
      calculate left edge
```

- 他ランクに読み書きを許す領域を登録
- 全ランクからの読み書き許可
- それまでの読み書きの完了確認
- 左ランクからの「上書き許可」確認
- データ書き込み
- 左ランクに「書き込み完了」通知
- 左ランクから「書き込み完了」確認
- 左ランクに「上書き許可」通知

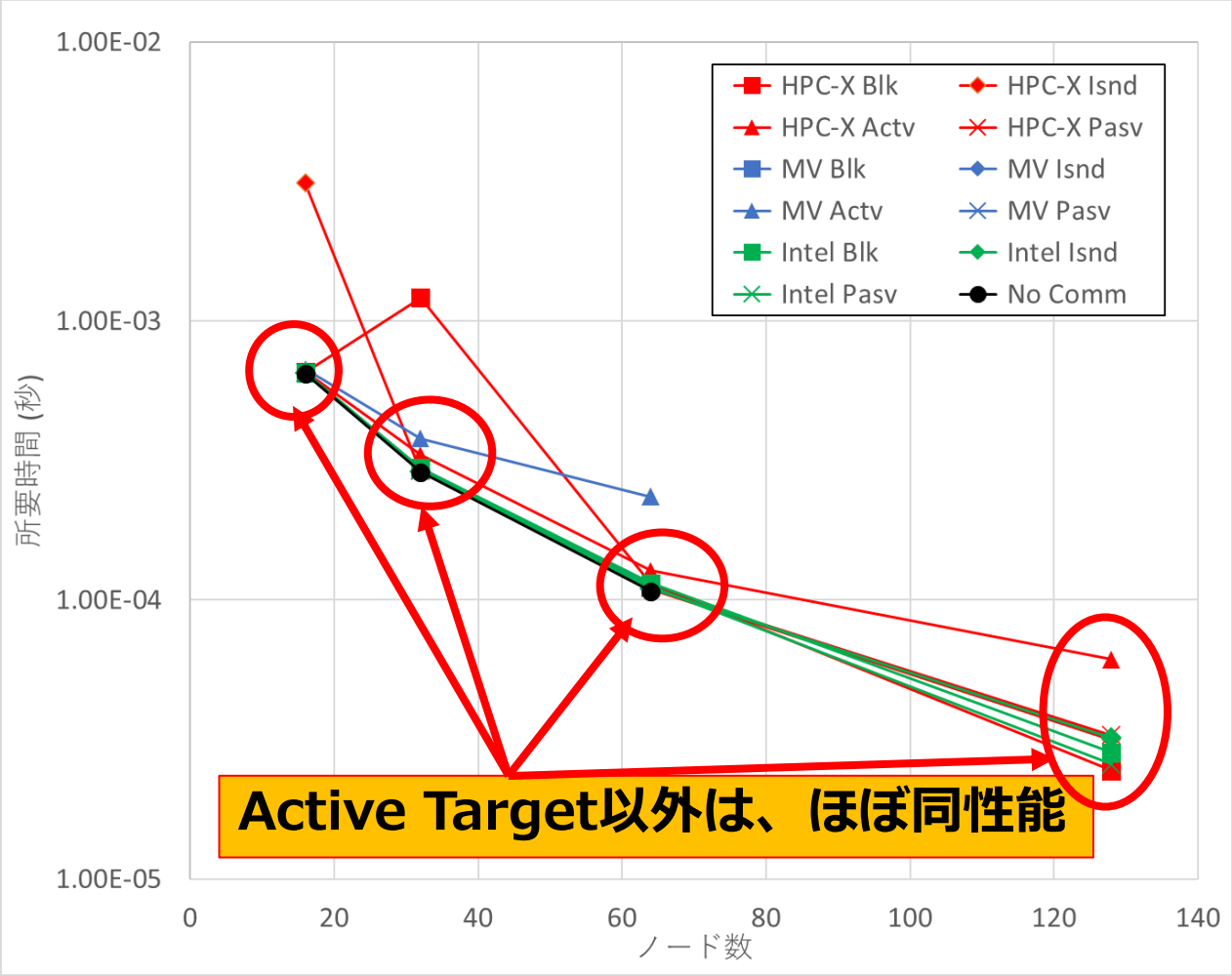


実験

- 計算環境
  - スーパーコンピュータシステム ITO
    - 九州大学情報基盤研究開発センター
    - Intel Xeon Gold 6154 (3.0GHz, 18core) x 2 / node
    - Mellanox InfiniBand EDR
    - Red Hat Linux Enterprise 7.3
    - gcc 4.8.5
- MPIライブラリ
  - HPC-X: Mellanox HPC-X 2.1.0
    - Mellanox社の高性能通信ツール群
      - SHARPライブラリ
      - MPIライブラリ (Open MPI 3.1.0 rc2 ベース)
  - MV: MVAPICH2 2.3.4
  - Intel: Intel MPI 2020 update 1
- 通信手段
  - Blk: 通信隠蔽無し
  - Isnd: MPI_Isend / Irecv
  - Actv: Active Target
  - Pasv: Passive Target
  - No Comm: 通信無し（参考）

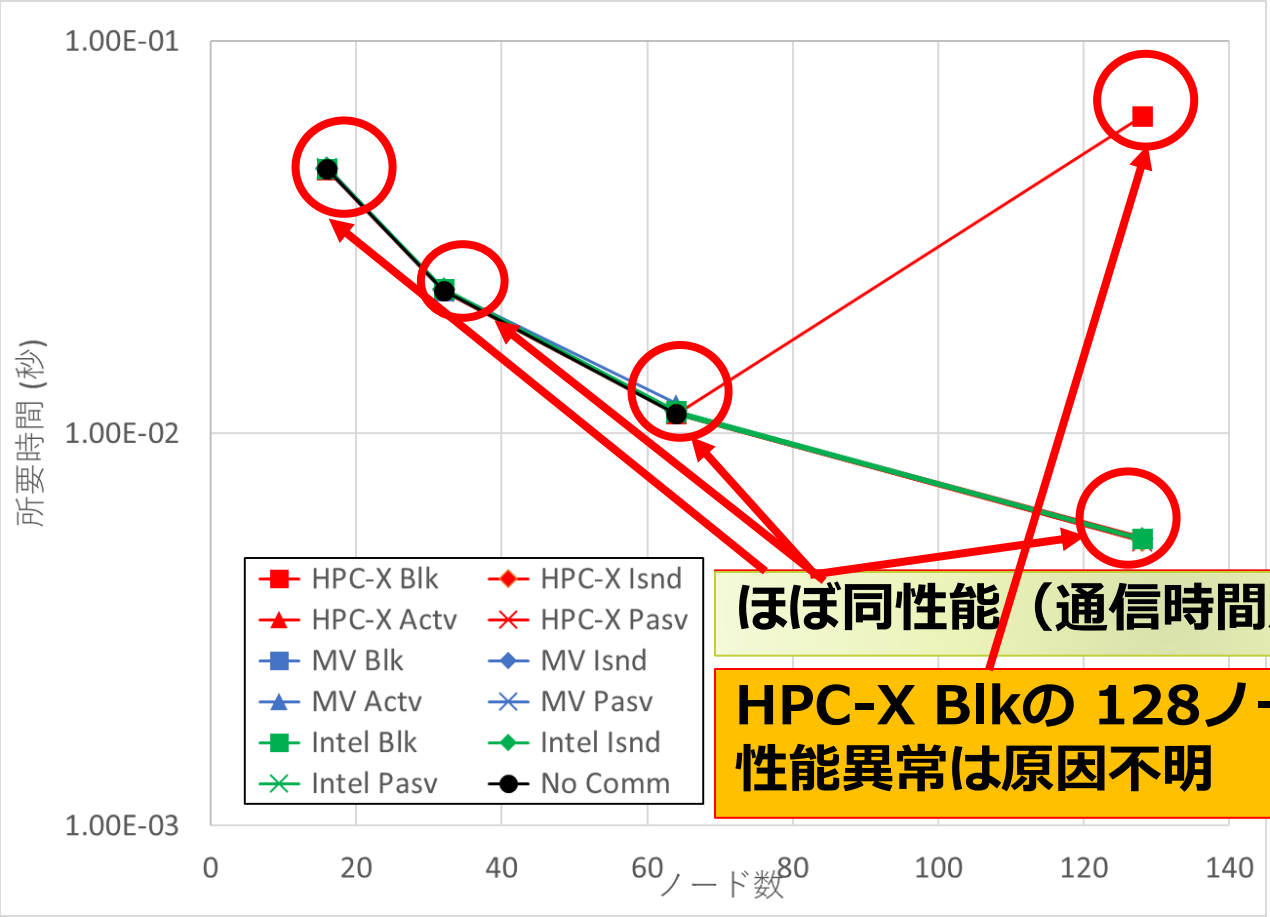
Stencil計算の所要時間

- 行列サイズ 1024 x 1024



Stencil計算の所要時間

- 行列サイズ 8192 x 8192

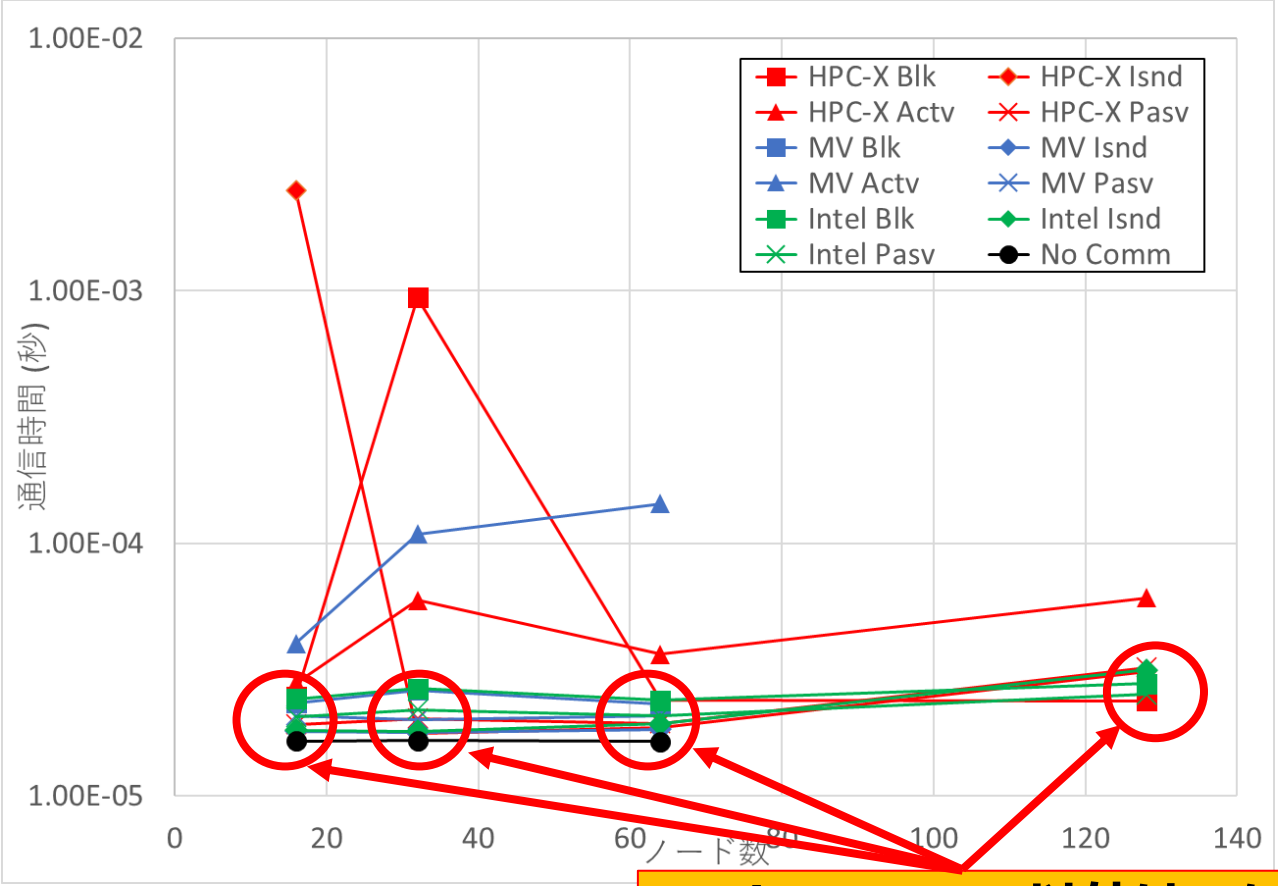


ほぼ同性能（通信時間が短すぎた）

HPC-X Blkの 128ノードでの性能異常は原因不明

Stencil計算内の隠蔽できなかった通信時間

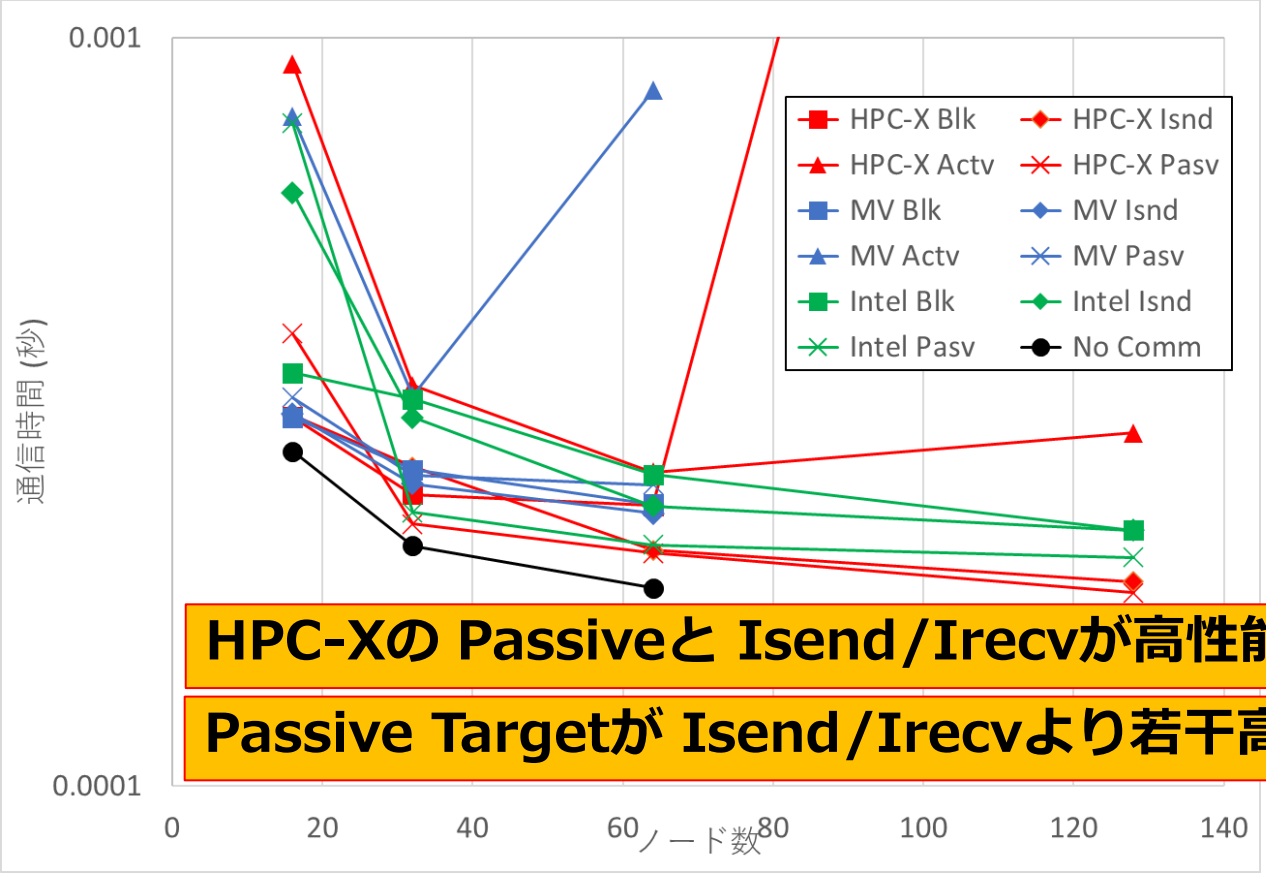
- 行列サイズ 1024 x 1024



Active Target以外は、ほぼ同性能

Stencil計算内の隠蔽できなかった通信時間

- 行列サイズ 8192 x 8192



HPC-Xの Passiveと Isend/Irecvが高性能

Passive Targetが Isend/Irecvより若干高速

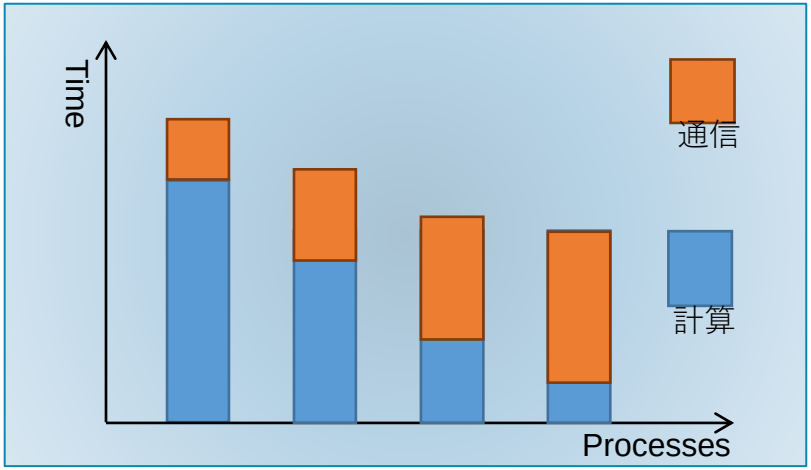
一対一通信の隠蔽効果

- 通信時間
  - Active Targetは逆効果
    - Intelは Active Targetで Segmentation Fault発生（原因調査中）
  - Passive Targetは Isend/Irecvと同等か、若干高速
    - 特に 8192x8192で 64ノード、128ノードの HPC-X、Intel
  - HPC-X、Intel は Isend/Irecv、Passive Targetで良好な性能
  - MVAPICH2 は通信隠蔽無しでは高速
    - 128ノードではエラー発生（原因調査中）

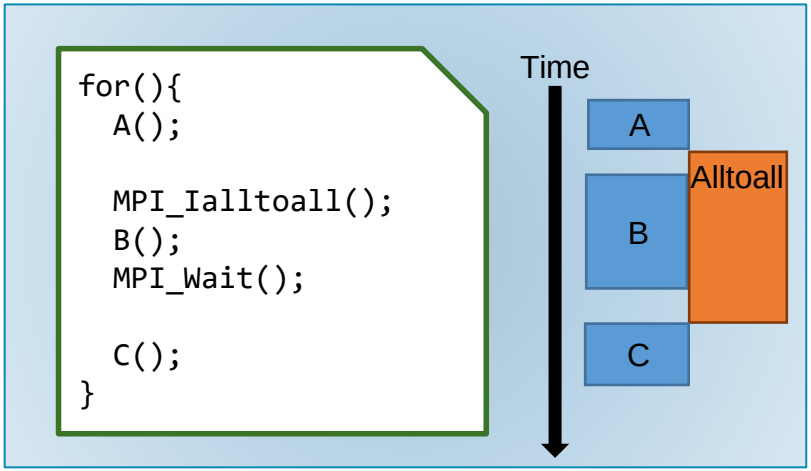
- 全体の所要時間
  - 今回の実験では、通信隠蔽の有無での有意な差はほとんど無し
    - 通信隠蔽効果が、非ブロッキング通信のオーバヘッドで相殺
    - より通信時間の比率が高い状況で検証したい

非ブロッキング集団通信

- 集団通信の所要時間
  - プロセス数に応じて増
  - プログラム全体の並列化効果さらに低減
- 非ブロッキング集団通信への期待
  - 通信隠蔽による並列化効果向上
- 非ブロッキング集団通信の課題
  - 集団通信アルゴリズムの推進手段

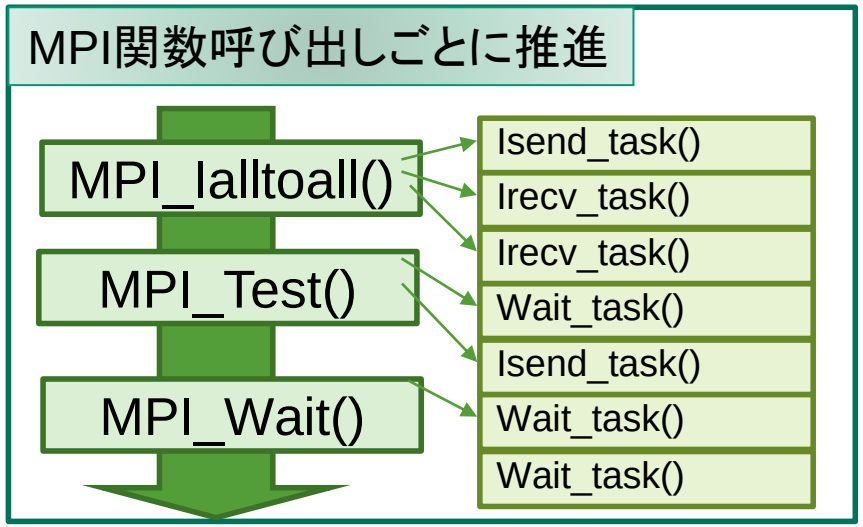


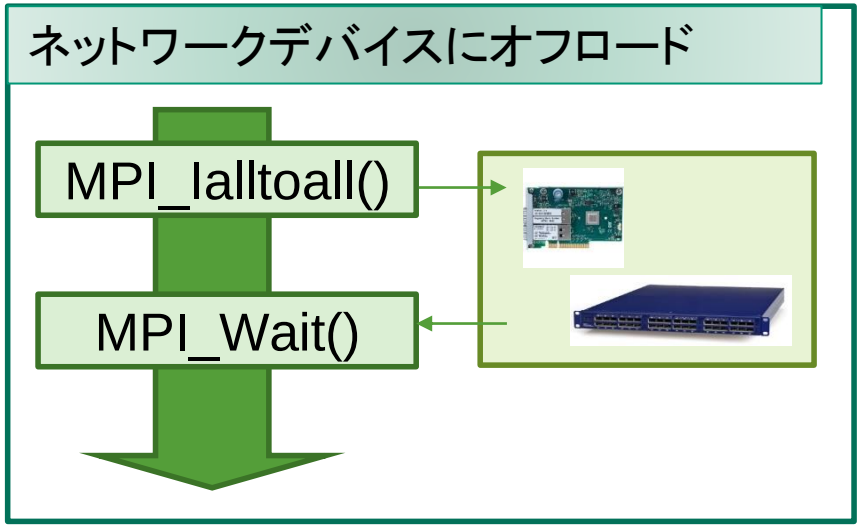
プロセス数の増加に伴う並列プログラムの実行時間の変化

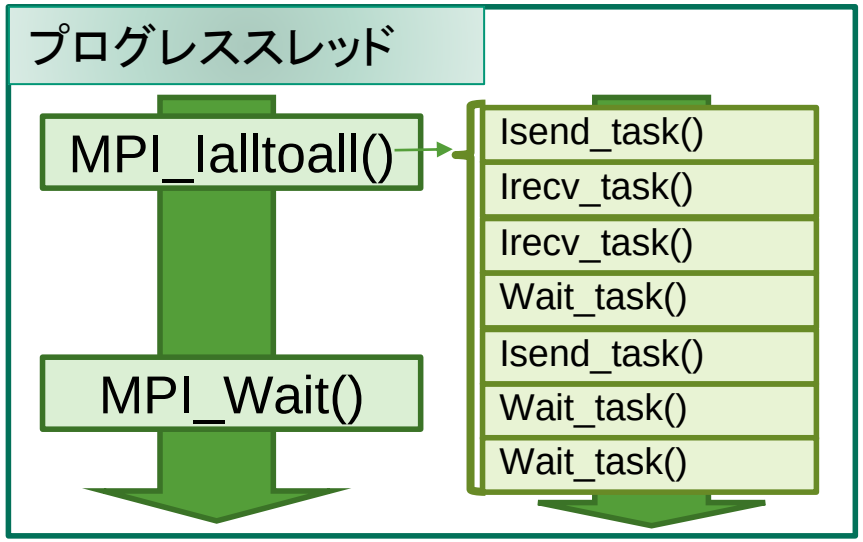


非ブロッキング集団通信による通信隠蔽

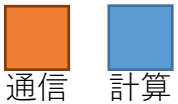
集団通信アルゴリズムの推進手段



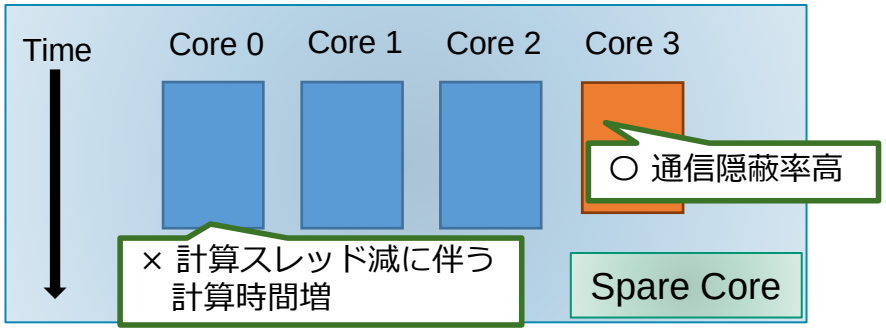




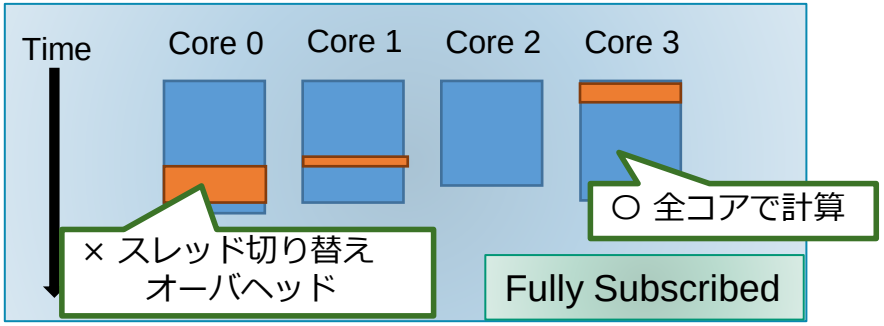
プログレススレッドによる推進



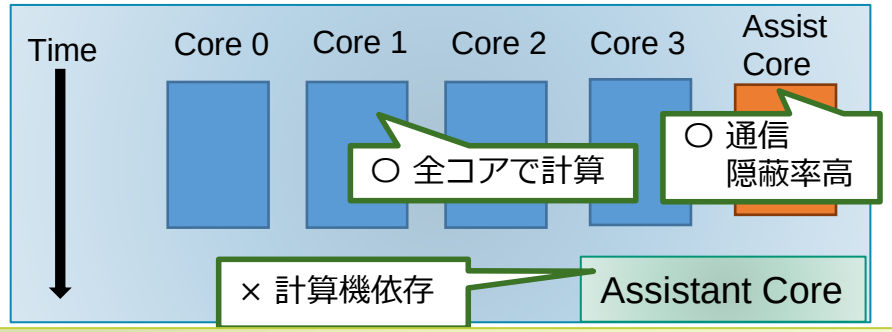
- Spare Core
  - プログレススレッド用に1コアを留保



- Fully Subscribed
  - 計算スレッドとコア共有



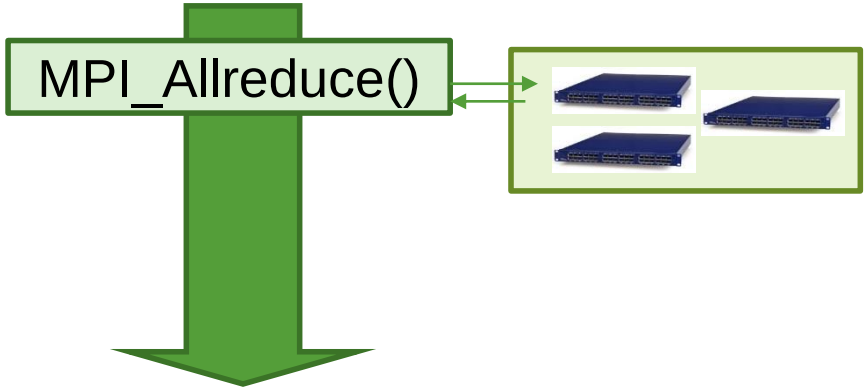
- Assistant Core
  - 計算機が提供する専用コアを使用

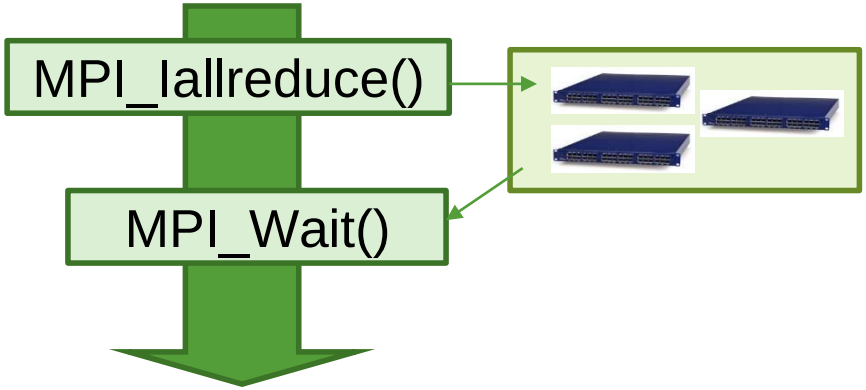


プロセス毎に 1つ多くスレッドを起動 ⇒ Assistant Core以外は計算性能に影響

オフロードによる推進 Mellanox SHARP

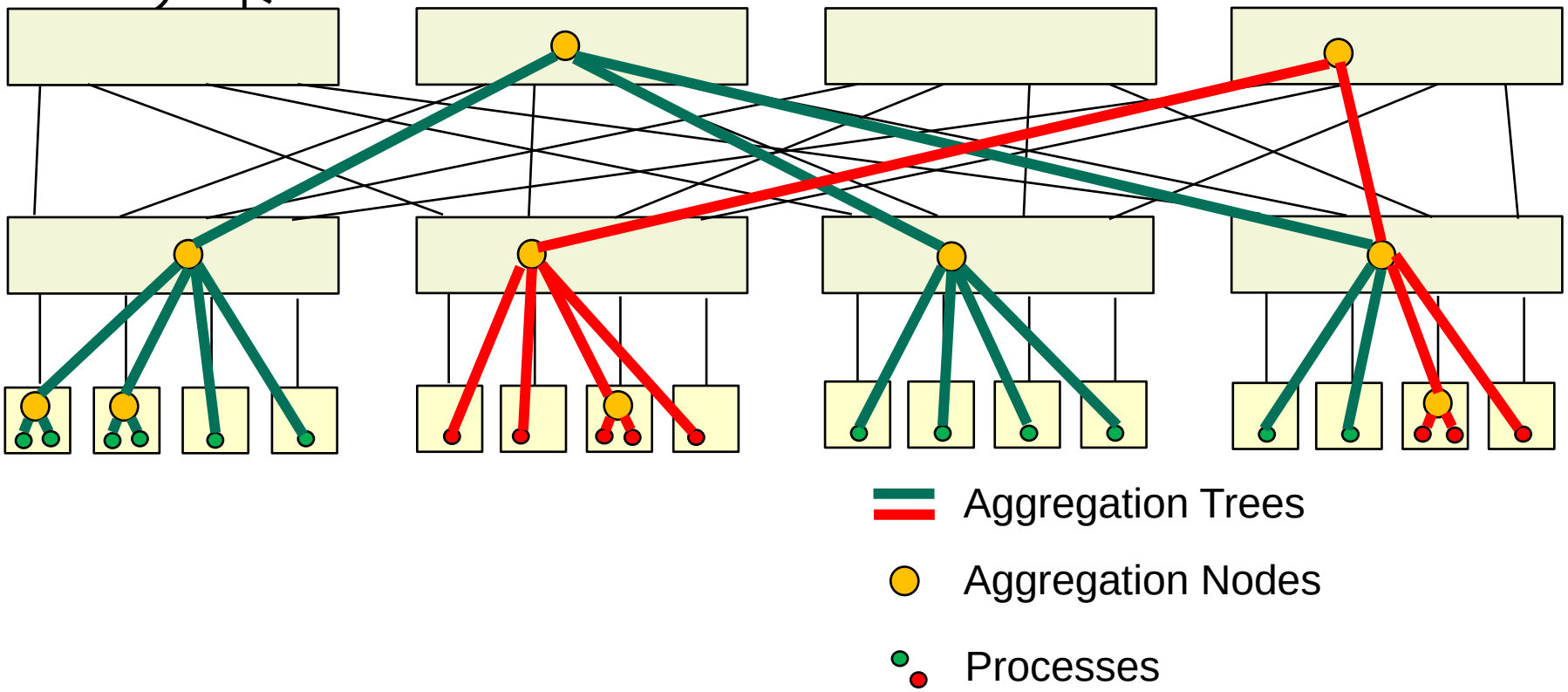
- InfiniBandスイッチに「集約型」の集団通信をオフロード
  - 主に、全プロセスのデータ集計（Allreduce）に利用
  - MPI_Allreduce, MPI_Iallreduceの両方に利用





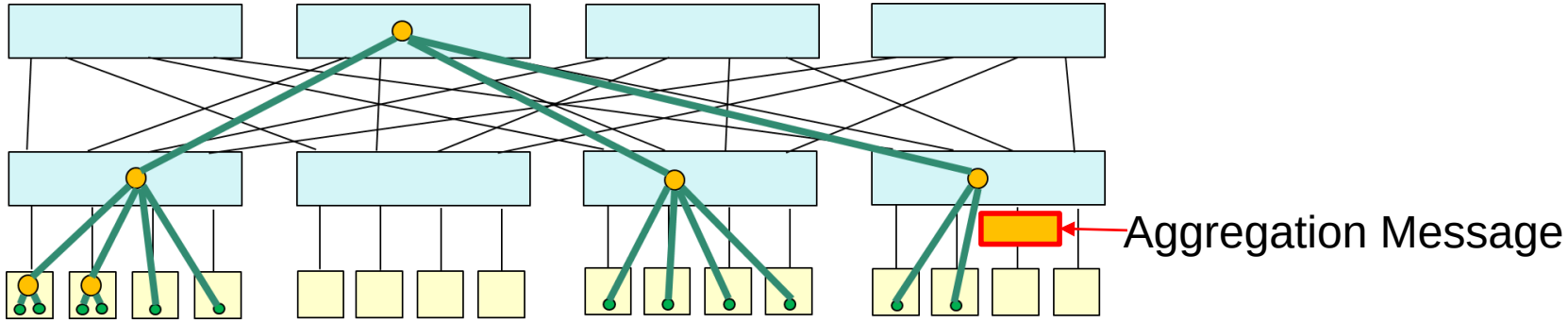
Aggregation Tree

- SHARPが Aggregation Nodeとプロセスで構成する仮想的な木構造
  - Aggregation Node: スイッチや計算ノードに配置する仮想的なノード



SHARPによる Allreduce

- Aggregation Tree上で Aggregation Messageにより上方へ集約し、結果を下方に伝搬
  - Aggregation Message:
    最大 256バイトのペイロードを持つ集約用メッセージ
    - 256バイトを超えるサイズ： 複数のメッセージに分割
  - 各 Aggregation Nodeで、集約演算の計算順序を保証



SHARPへの期待

今回検証

- Allreduce, Iallreduceの所要時間短縮 ○
  - データ転送経路上で演算するためデータ転送量を削減
- 高い通信隠蔽率 ○
- OSジッタ等、プロセス間の不均衡な処理による影響軽減
  - CPUにおける待ち時間が不要

非ブロッキング集団通信の効果計測例：OSU Benchmark

- オーバラップ率重視
  - 実行毎に通信時間を計測し、ちょうど隠蔽できるように計算量を調整

- 実験毎に計算量が変動

- 実装手段の相互評価が困難
- 計算まで含めた全体的な効果の評価が困難

```
ts = MPI_Wtime();
MPI_Ialltoall();
MPI_Wait();
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Ialltoall();
tcs = MPI_Wtime();
while (MPI_Wtime() - tcs < Tcomm)
    dummy_comp();
Tcomp = MPI_Wtime() - tcs;
MPI_Wait();
Tall = MPI_Wtime() - ts;

overlap = (Tall - Tcomp) / Tcomm;
```

通信時間

通信時間を超えるまで計算

計算時間

全体時間

今回使用したプログラム

- 通信量、計算量は
  実行時にパラメータ指定

- 同じパラメータによる全体時間で実装手段の優劣を評価

- 実装手法相互の比較
- 全体的な効果の評価

```
get_param(&M, &N);

ts = MPI_Wtime();
MPI_Allreduce(M);
Tcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Iallreduce (M);
MPI_Wait();
Tnbcomm = MPI_Wtime() - ts;

ts = MPI_Wtime();
do_comp(N);
Tcomp = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Allreduce (M);
do_comp(N);
Tblock = MPI_Wtime() - ts;

ts = MPI_Wtime();
MPI_Iallreduce (M);
do_comp(N);
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

ブロッキング集団通信

非ブロッキング集団通信

計算

ブロッキング通信 + 計算

非ブロッキング通信 + 計算

計算コード

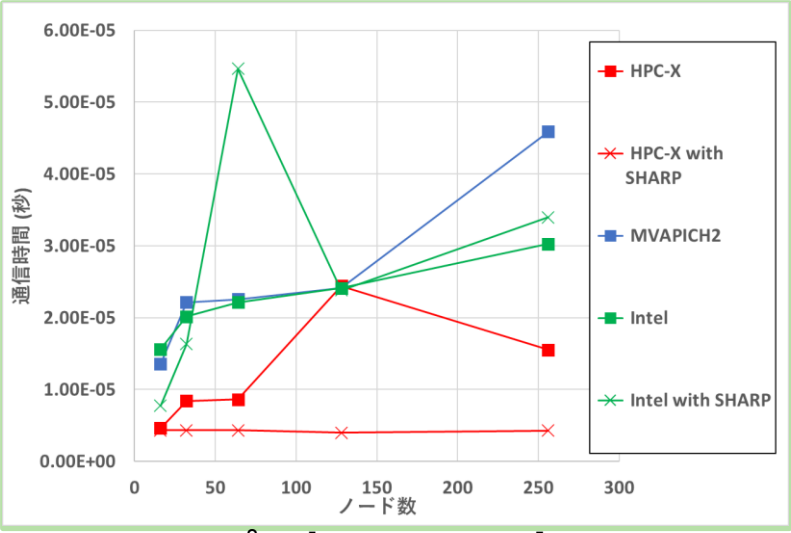
- NxN密行列積
  - 全プロセスで、同じサイズの行列積を計算
    - 各プロセス内でスレッド並列：
      - 最外ループを並列化

```
do_comp(N)
{
  #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++)
          for (k = 0; k < N; k++)
              for (j = 0; j < N; j++)
                  c[i*N+j] += a[i*N+k] * b[k*N+j];
}
```

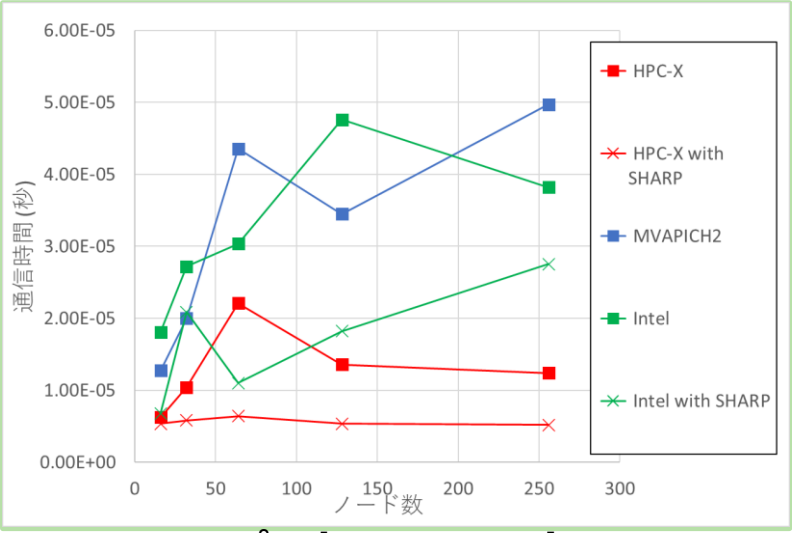
実験

- 計算環境
  - スーパーコンピュータシステム ITO
    - 九州大学情報基盤研究開発センター
    - Intel Xeon Gold 6154 (3.0GHz, 18core) x 2 / node
    - Mellanox InfiniBand EDR
    - Red Hat Linux Enterprise 7.3
    - gcc 4.8.5
- MPIライブラリ
  - Mellanox HPC-X 2.1.0
    - SHARP有効化：mpirunに以下のオプション追加
      - -x HCOLL_ENABLE_SHARP=2 -x HCOLL_ENABLE_NBC=1 -x HCOLL_POLLING_LEVEL=1
  - MVAPICH2 2.3.4
    - SHARP有効化：以下の環境変数を設定
      - export MV2_ENABLE_SHARP=1
    - 内部で HPC-Xを利用
      - ただし、サポート対象は HPC-X 1.7.0および HPC-X 1.8.0
  - Intel MPI 2020 update 1
    - SHARP有効化：以下の環境変数を設定
      - export I_MPI_ADJUST_ALLREDUCE=24
      - export I_MPI_ADJUST_IALLREDUCE=9
      - export I_MPI_COLL_EXTERNAL=1

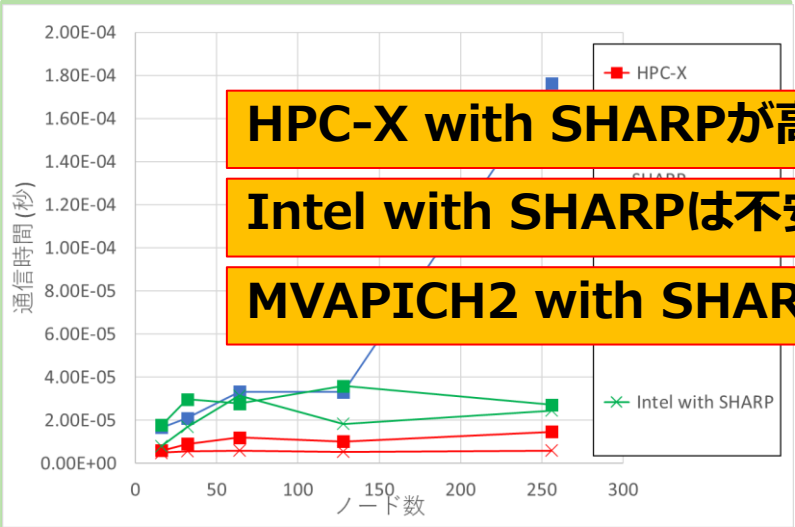
MPI_Allreduce 通信時間



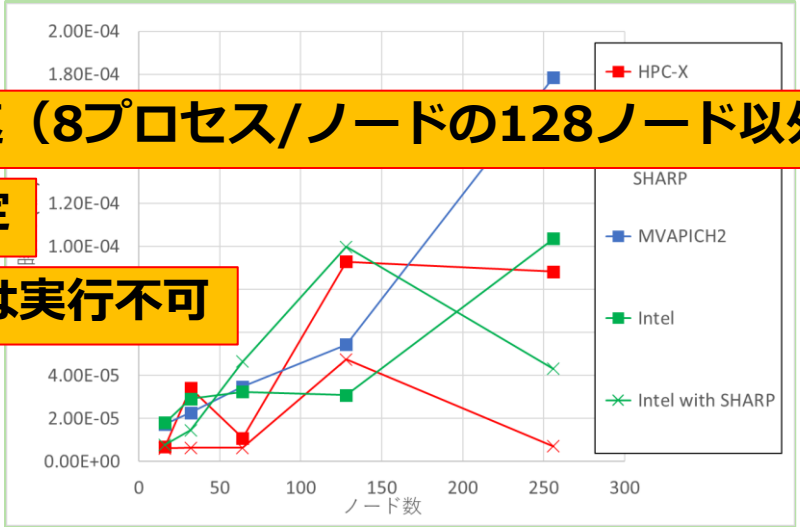
1プロセス / ノード



2プロセス / ノード



4プロセス / ノード



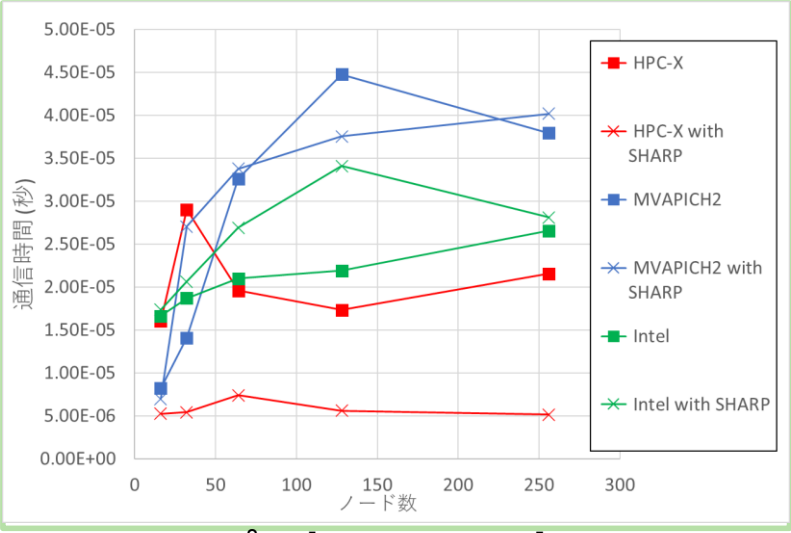
8プロセス / ノード

HPC-X with SHARPが高速（8プロセス/ノードの128ノード以外）

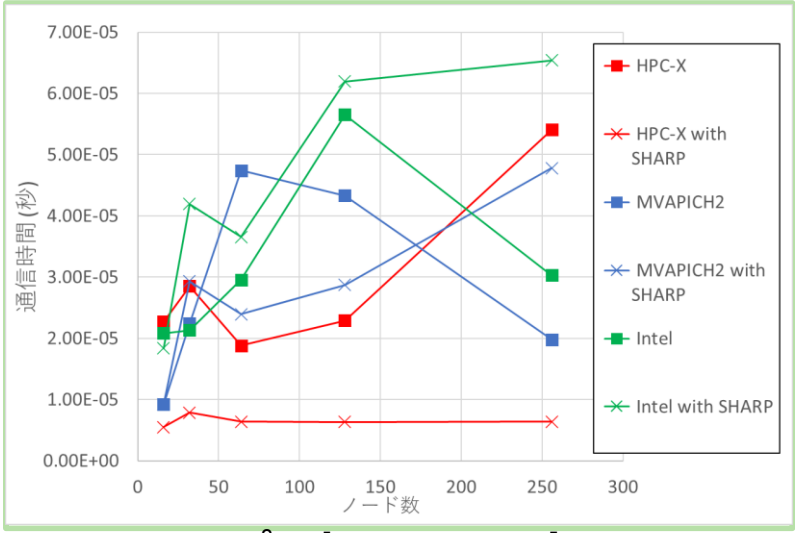
Intel with SHARPは不安定

MVAPICH2 with SHARPは実行不可

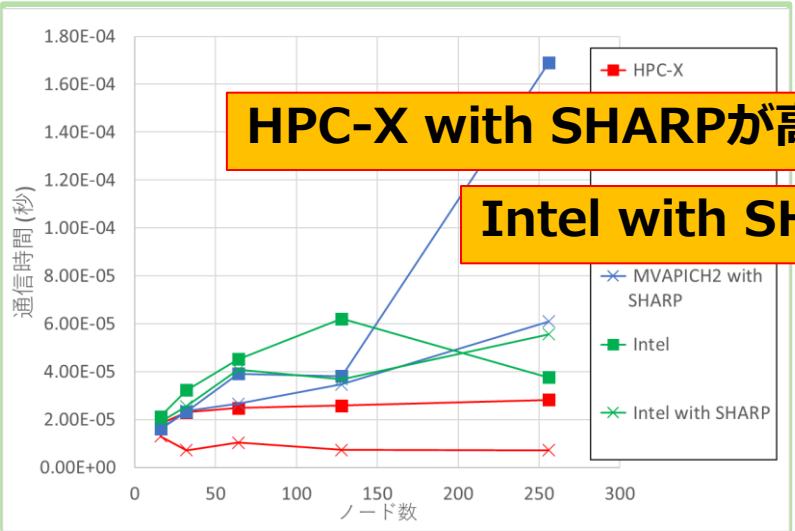
MPI_Iallreduce + Wait 通信時間



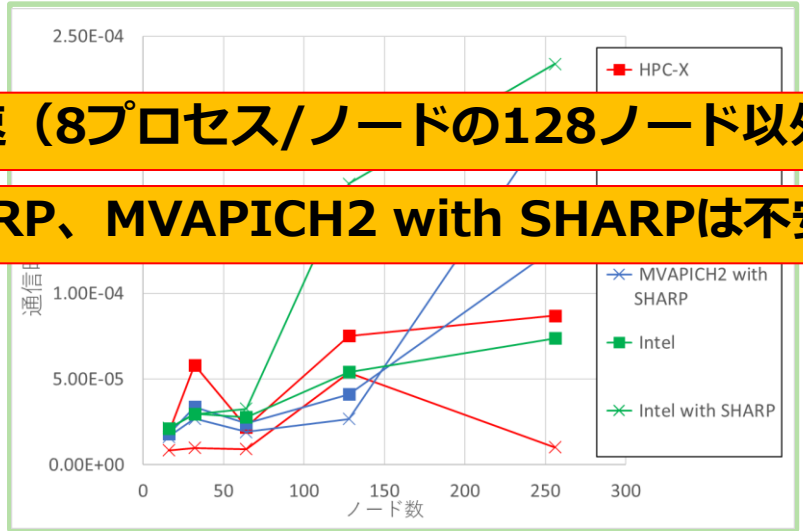
1プロセス / ノード



2プロセス / ノード



4プロセス / ノード



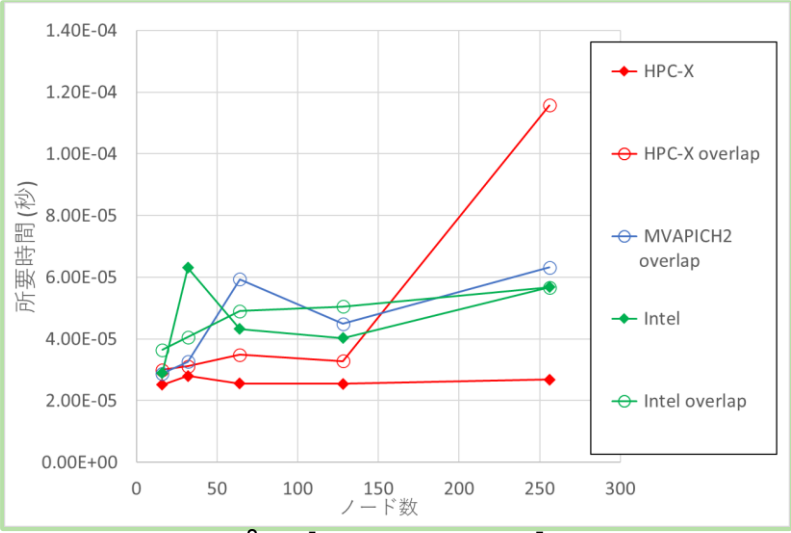
8プロセス / ノード

HPC-X with SHARPが高速（8プロセス/ノードの128ノード以外）

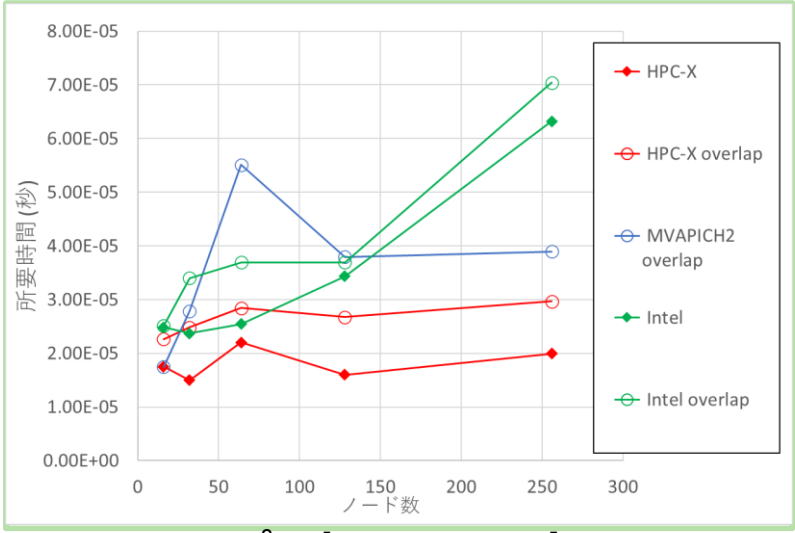
Intel with SHARP、MVAPICH2 with SHARPは不安定

計算＋通信の所要時間

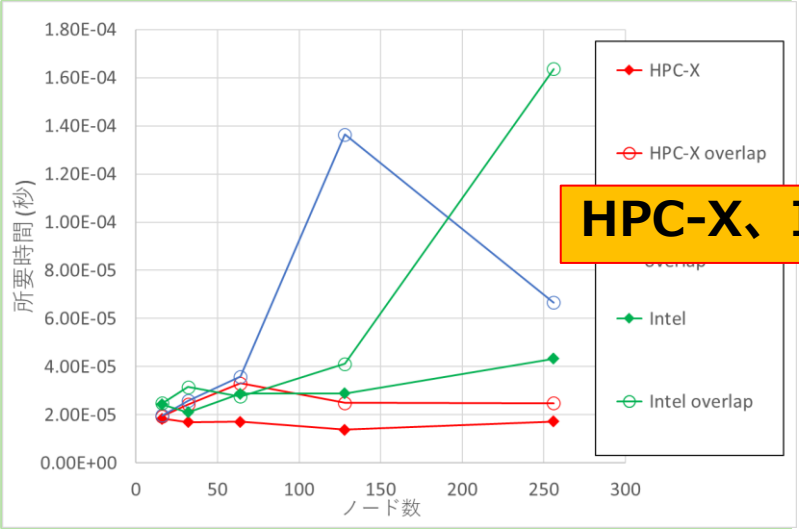
計算：32x32の行列積



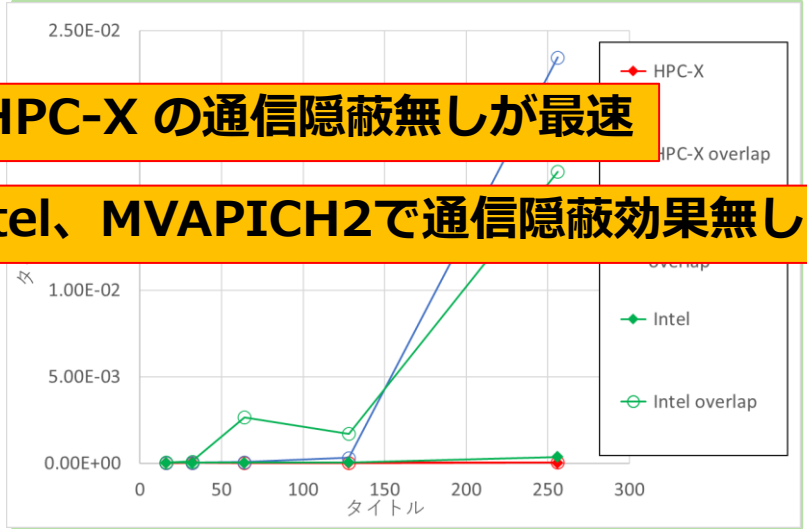
1プロセス / ノード



2プロセス / ノード



4プロセス / ノード



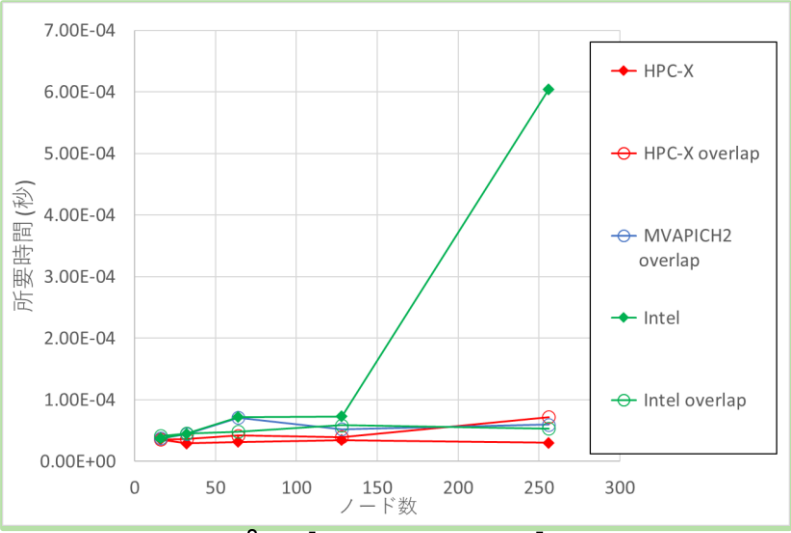
8プロセス / ノード

HPC-X の通信隠蔽無しが最速

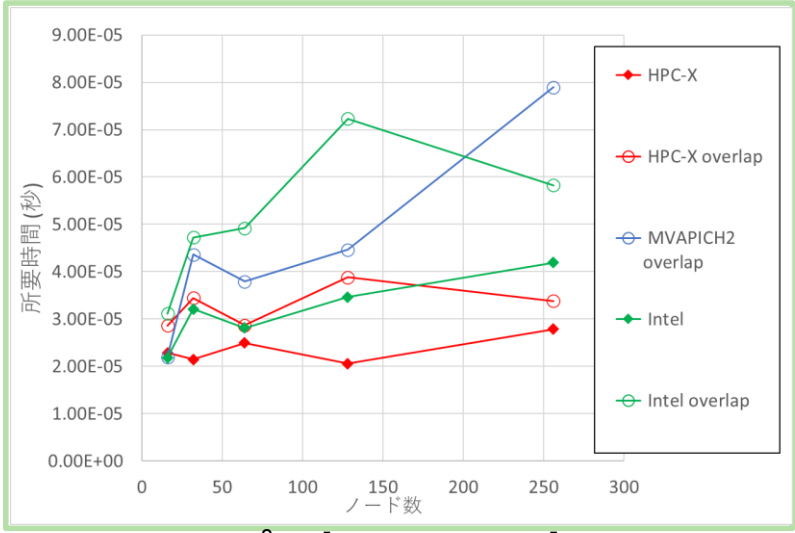
HPC-X、Intel、MVAPICH2で通信隠蔽効果無し

計算＋通信の所要時間

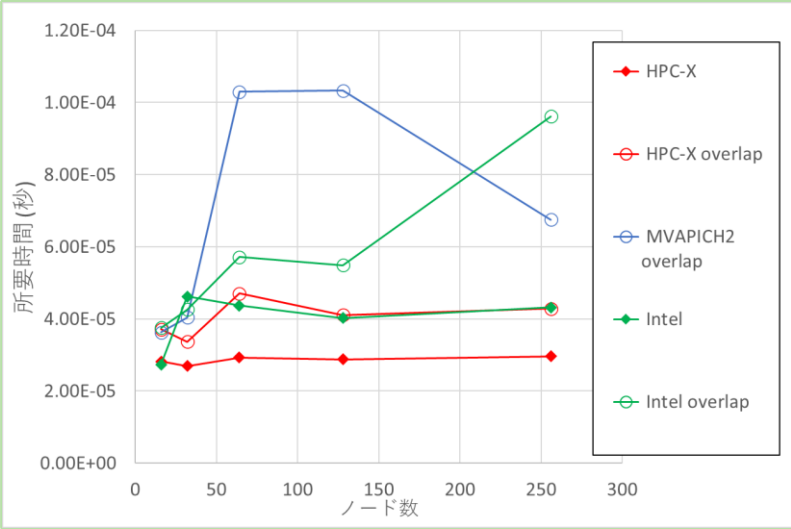
計算：64x64の行列積



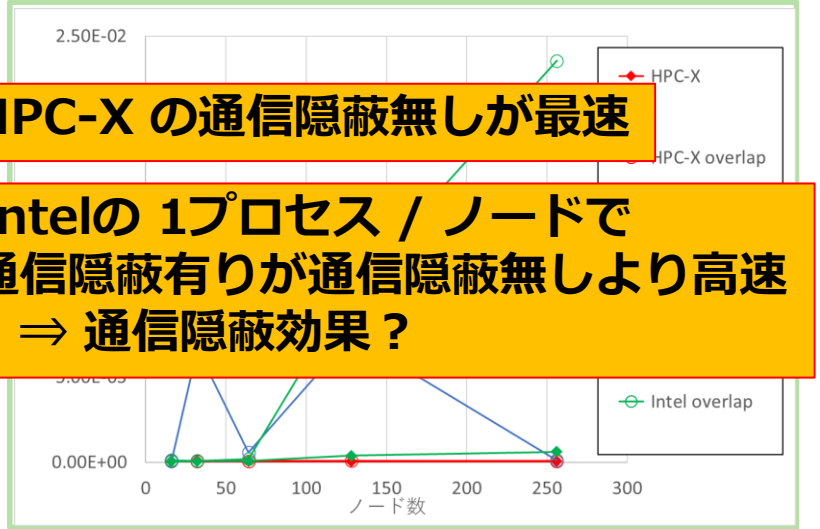
1プロセス / ノード



2プロセス / ノード



4プロセス / ノード



8プロセス / ノード

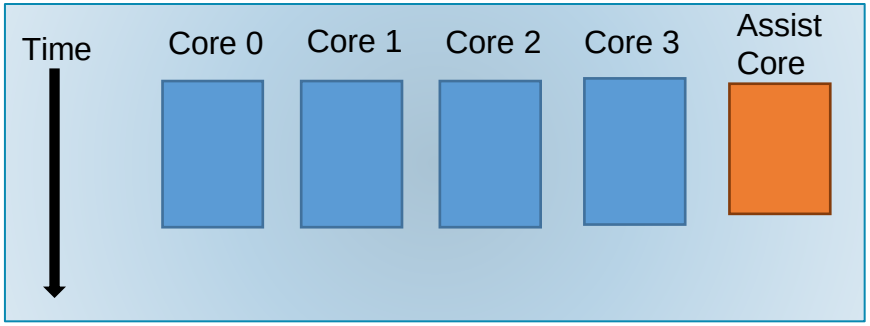
HPC-X の通信隠蔽無しが最速

Intelの 1プロセス / ノードで
通信隠蔽有りが通信隠蔽無しより高速
⇒ 通信隠蔽効果？

(参考)
Fujitsu FX100の Assistant Core

- 利用方法
  - 実行時オプション
    - モード番号：
      - 1: 指定区間（MPI呼び出し無し）
      - ２： 指定区間（MPI呼び出し有り）
      - ３： 自動区間

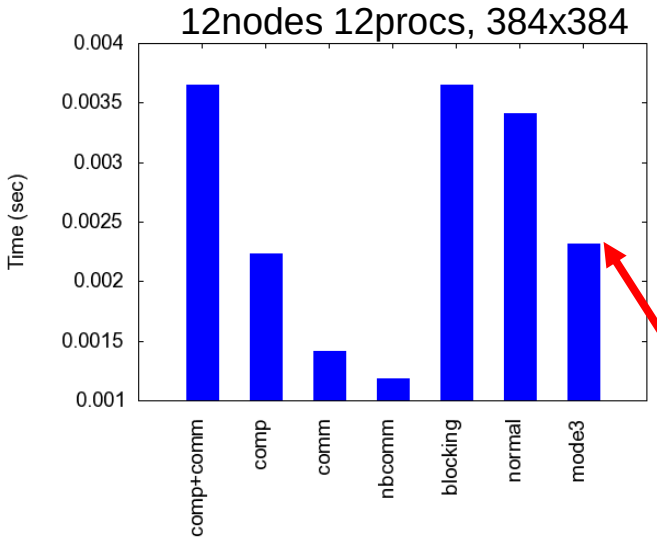
```
--mca opal_progress_thread_mode モード番号
```

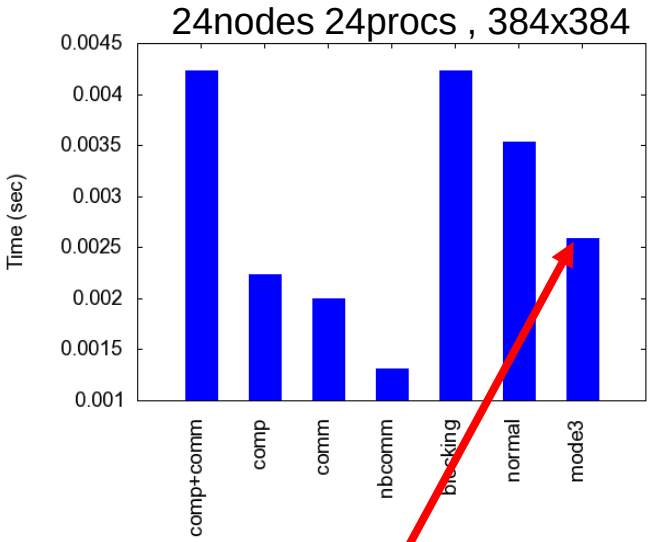


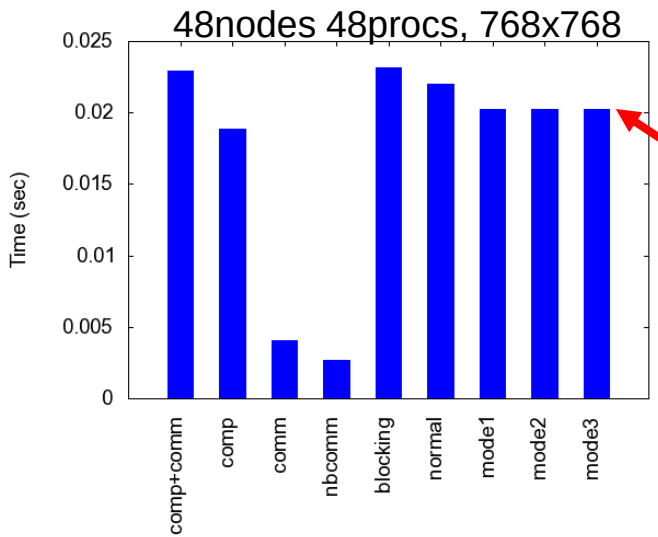
  - 区間指定
    - モード番号 1, 2のみ

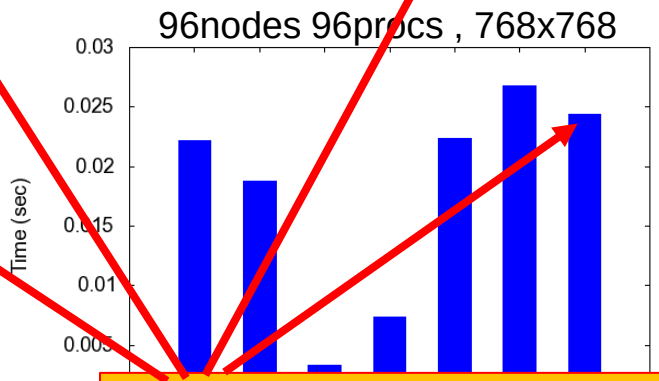
```
ts = MPI_Wtime();
MPI_Ialltoall(M);
FJMPI_Progress_start();
do_comp(N);
FJMPI_Progress_stop();
MPI_Wait();
Tovlp = MPI_Wtime() - ts;
```

FX100における通信隠蔽効果（1プロセス / ノード）



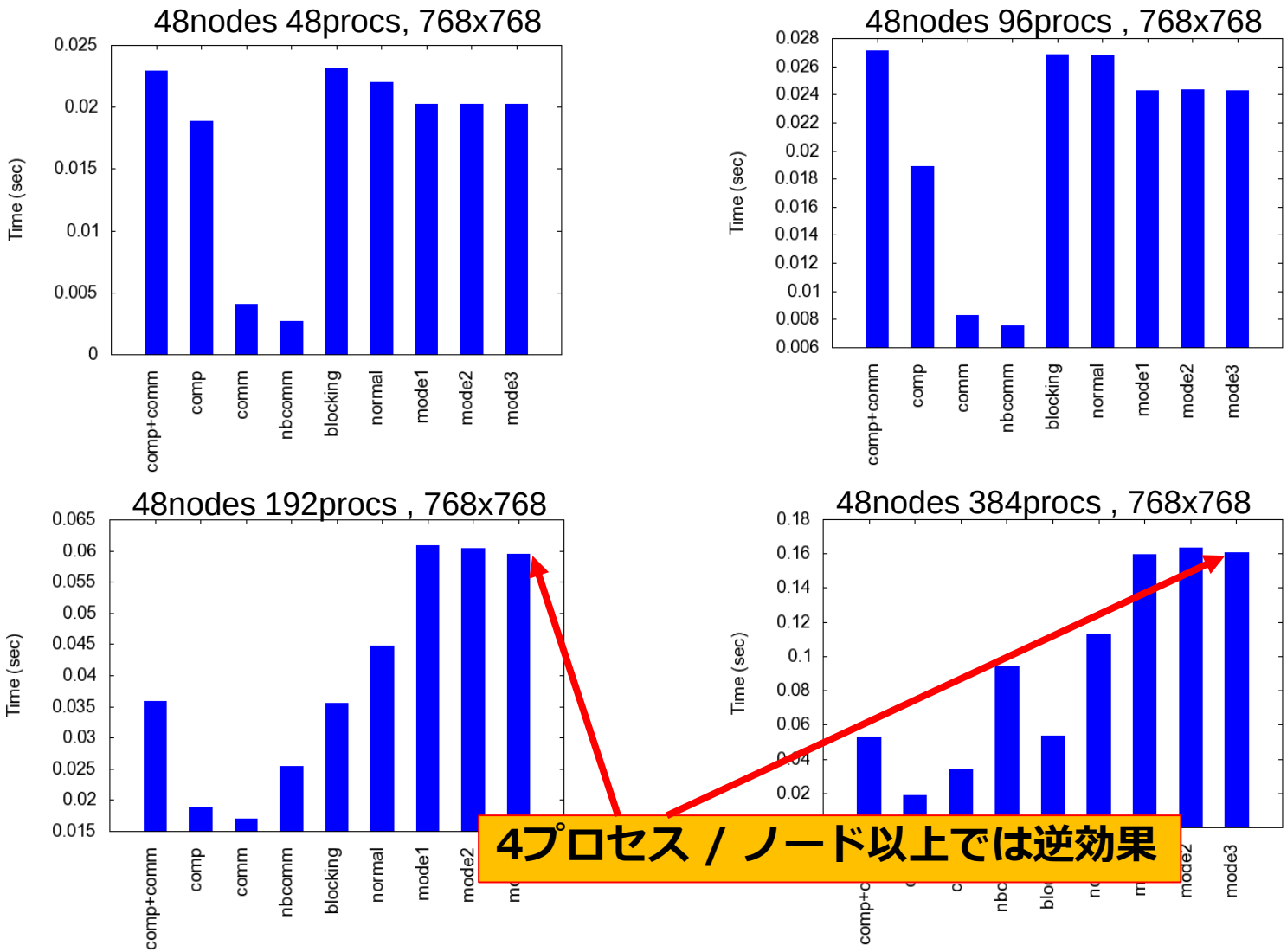






通信隠蔽効果あり（blockingやcomp+commより高速）

FX100における通信隠蔽効果



まとめ

- 一対一通信の通信隠蔽効果
  - MPI_Isend/Irecv、片側通信とも、通信時間は削減できた
  - 全体の所要時間については、オーバヘッド等の影響で、今回は効果が確認できなかった
- 集団通信のオフロード
  - MPI_Allreduceの通信時間は、大幅な短縮を確認できた
    - HPC-X使用時
  - 通信隠蔽効果は、現時点では確認できなかった
- （参考）Assistant Core
  - FX100では、2プロセス / ノードまで通信隠蔽効果を確認できた
  - 富岳、不老での効果を期待