



# 筑波大学計算科学研究センター HPCセミナー 「最適化2」(通信最適化)

建部修見

tatebe@cs.tsukuba.ac.jp

筑波大学計算科学研究センター



# 講義内容

- 基本通信性能
  - 1対1通信
  - 集団通信
- プロファイラ
- 通信最適化
  - 通信の削減
  - 通信遅延隠蔽
  - 通信ブロック
  - 負荷分散



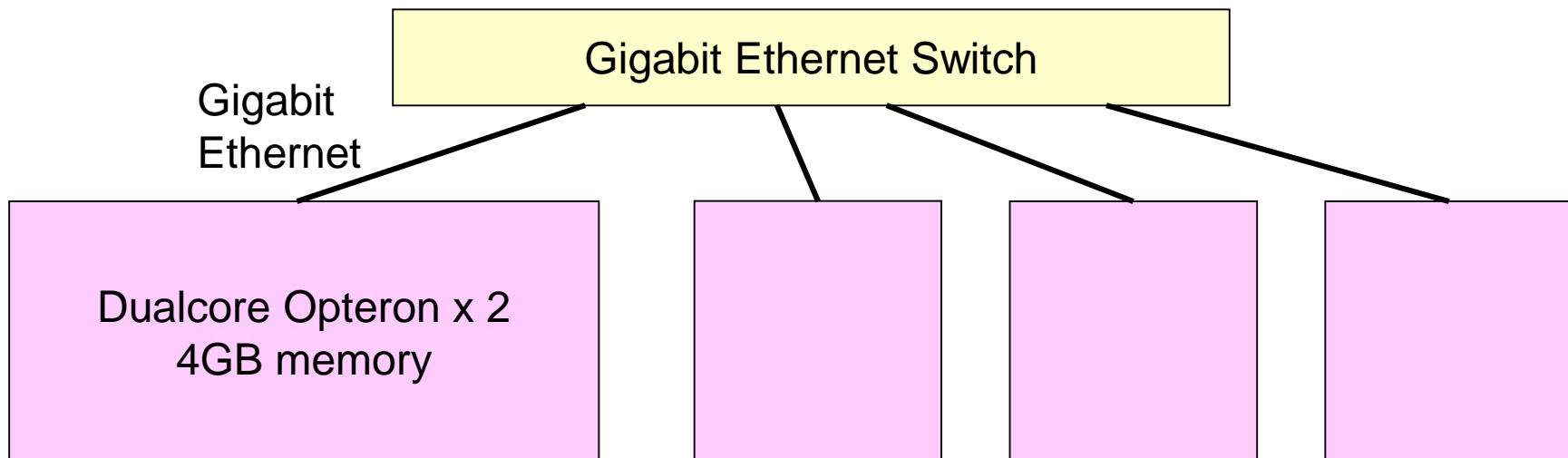
# 基本通信性能

- 通信最適化のためには基本通信性能を押さえておくことが重要！
  - 各種通信パターンにおける通信性能の把握
  - 通信ブロッキングのブロックサイズの決定
  - ネットワーク性能と比較して、通信ライブラリ自体の性能改善



# 基本通信性能評価環境(1)

- 4クラスタノード
  - 2.6GHz Dualcore Opteron x 2 sockets (4 cores)
  - 4GB memory
  - Linux 2.6.18-1.2798.fc6
  - OpenMPI 1.1-7.fc6
- Gigabit Ethernetで接続
  - TCPでの理論ピーク性能は949Mbps(=113.1MB/sec)





# 基本通信性能評価環境(2)

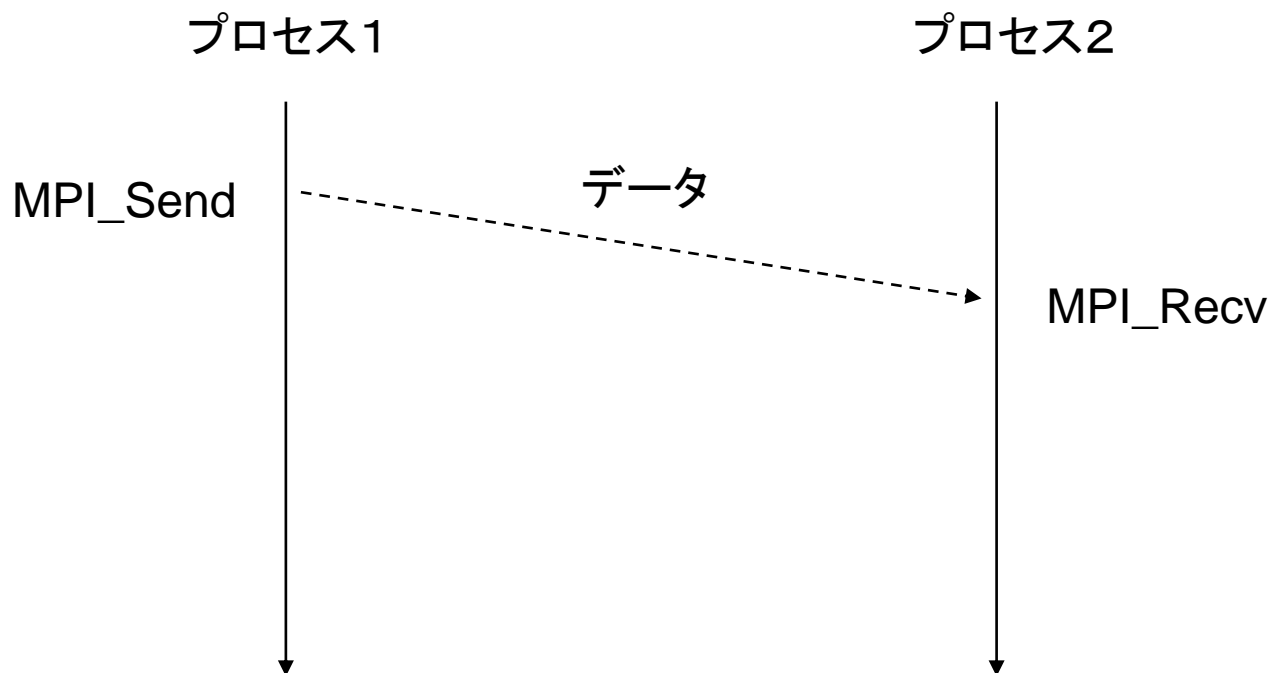
- Oakforest-PACS4ノード
  - 1.4GHz Xeon Phi (Knights Landing; KNL) (68 cores)
  - 96GB DDR4 + 16GB MCDRAM
  - Intel MPI
- Omni-Pathで接続
  - 物理性能は100 Gbps
- メモリ配置などの最適化は施さず





# 1対1通信の性能

- 1対1通信はMPIにおける基本通信







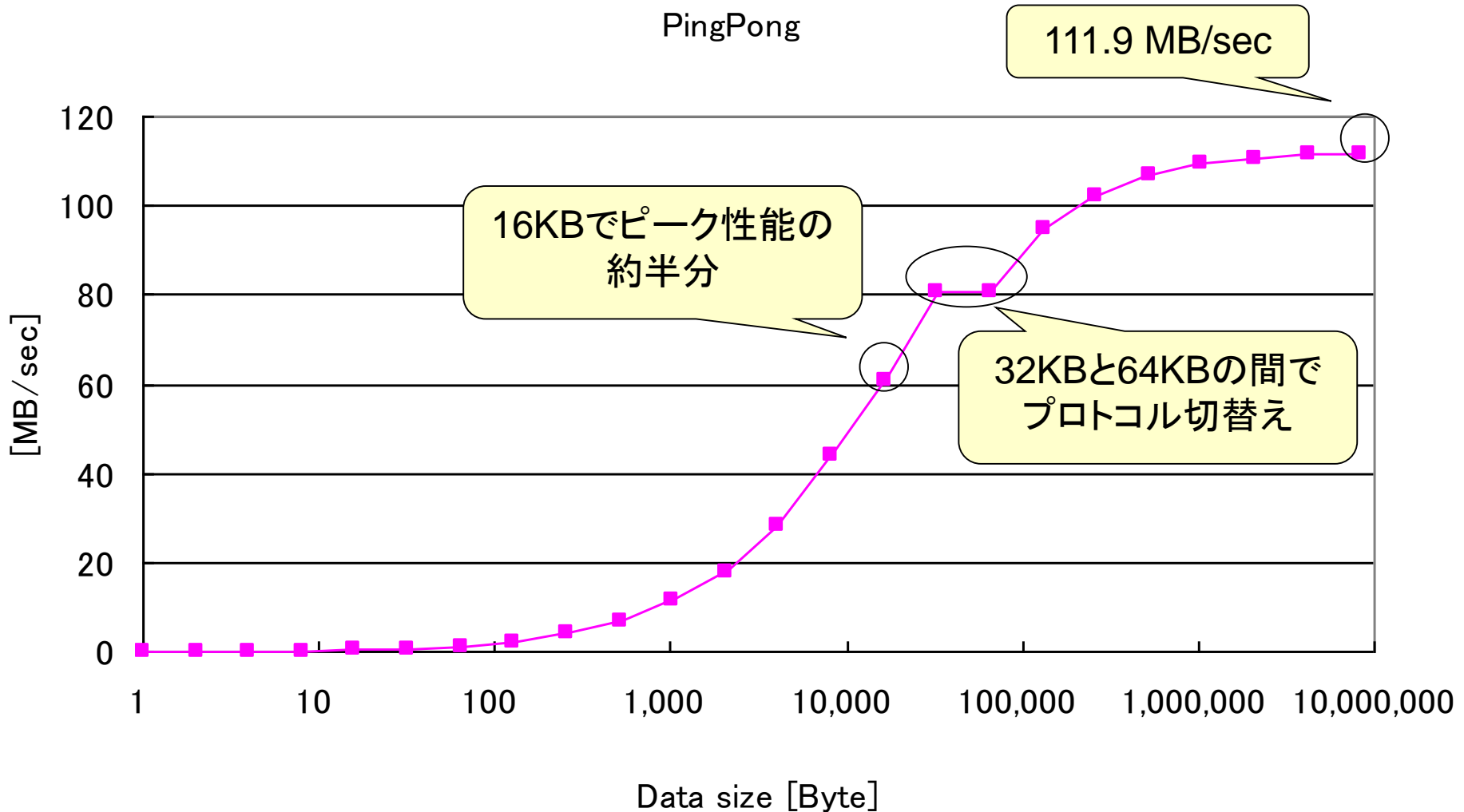
# PingPongベンチマークの例

```
for (s = 1; s <= P MAX_MSGSIZE; s <<= 1) {
  t = MPI_Wtime();
  for (i = 0; i < ITER; ++i)
    if (rank == 0) {
      MPI_Send(BUF, s, MPI_BYTE, 1, TAG1, COMM);
      MPI_Recv(BUF, s, MPI_BYTE, 1, TAG2, COMM, &status);
    } else if (rank == 1) {
      MPI_Recv(BUF, s, MPI_BYTE, 0, TAG1, COMM, &status);
      MPI_Send(BUF, s, MPI_BYTE, 0, TAG2, COMM);
    }
  t = (MPI_Wtime() - t) / 2 / ITER;
  if (rank == 0)
    printf(“%d %g %g¥n”, s, t, s / t); // サイズ、時間、バンド幅
}
```





# [環境 1] PingPongベンチマーク





# 1対1通信プロトコル

- Eagerプロトコル(1-wayプロトコル)
  - 短メッセージ
  - メッセージヘッダとデータ(ペイロード)を同時に送信
  - 低遅延だが受信側でコピーのオーバヘッドが発生
- ランデブ(Rendezvous)プロトコル(3-wayプロトコル)
  - 長メッセージ
  - メッセージヘッダを送信し、完了通知を待ち、データを送信
  - 高バンド幅だがeagerプロトコルに比べ高遅延



# 1対1通信プロトコル(続き)

- MPI処理系は、メッセージ長によりプロトコルを選択
- メッセージ長を変えて計測することにより明らかに
- プロトコル切替のメッセージ長は、通信性能最適化のために指定可能なことが多い



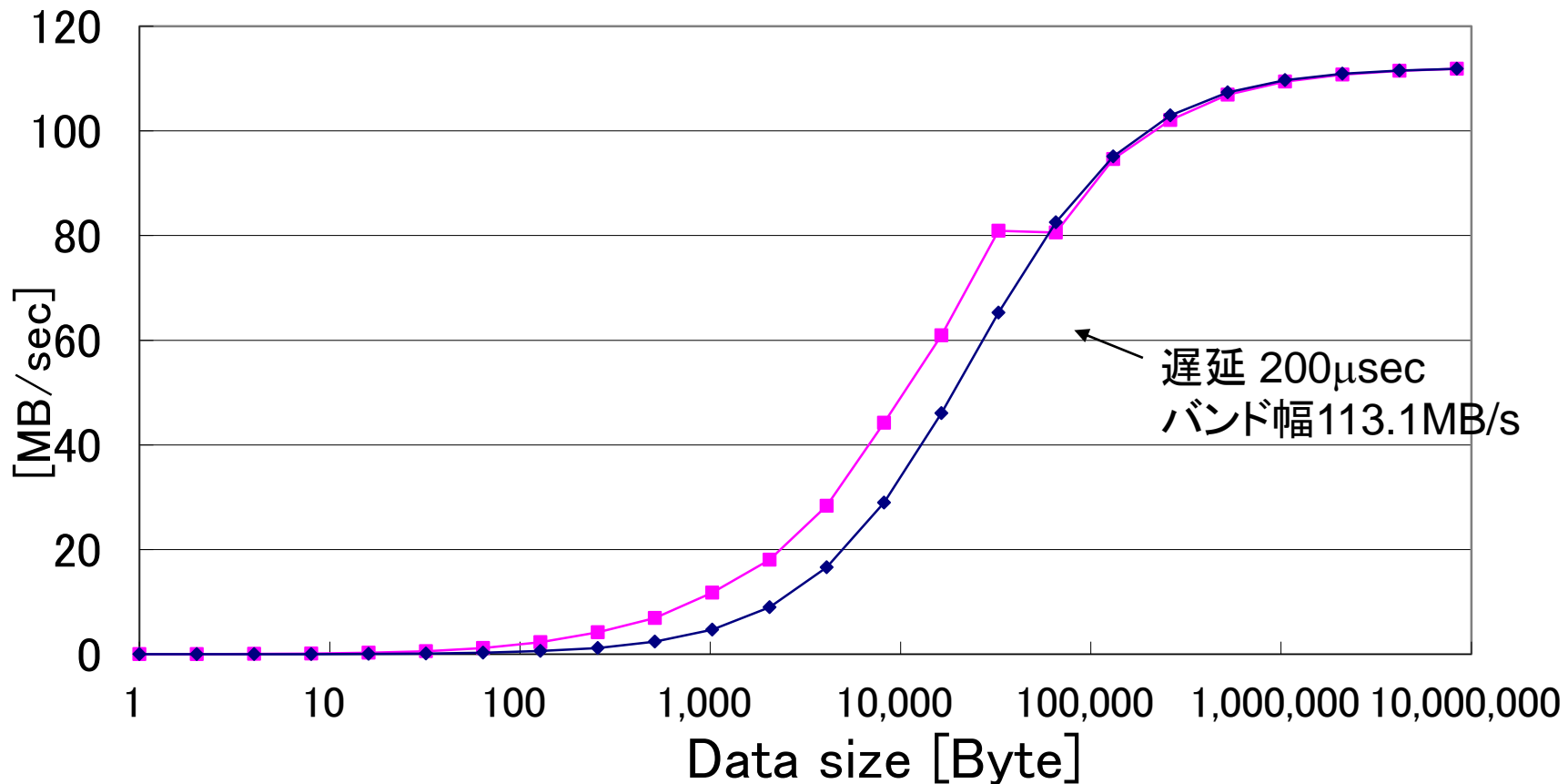
# 1対1通信の理論性能

- 遅延  $L$ 秒
- 最大バンド幅  $B$ バイト/秒
- $n$ バイト送信にかかる時間  $L + n/B$
- $n$ バイト送信のバンド幅  $n/(L + n/B)$
- 半分の性能を出すための転送サイズ  $n$

$$\frac{n}{L + \frac{n}{B}} = \frac{B}{2} \quad \text{これを解いて} \quad n_{half} = BL$$



# [環境1] 遅延、バンド幅での曲線と の比較



理論曲線  $n / (L + n / B)$

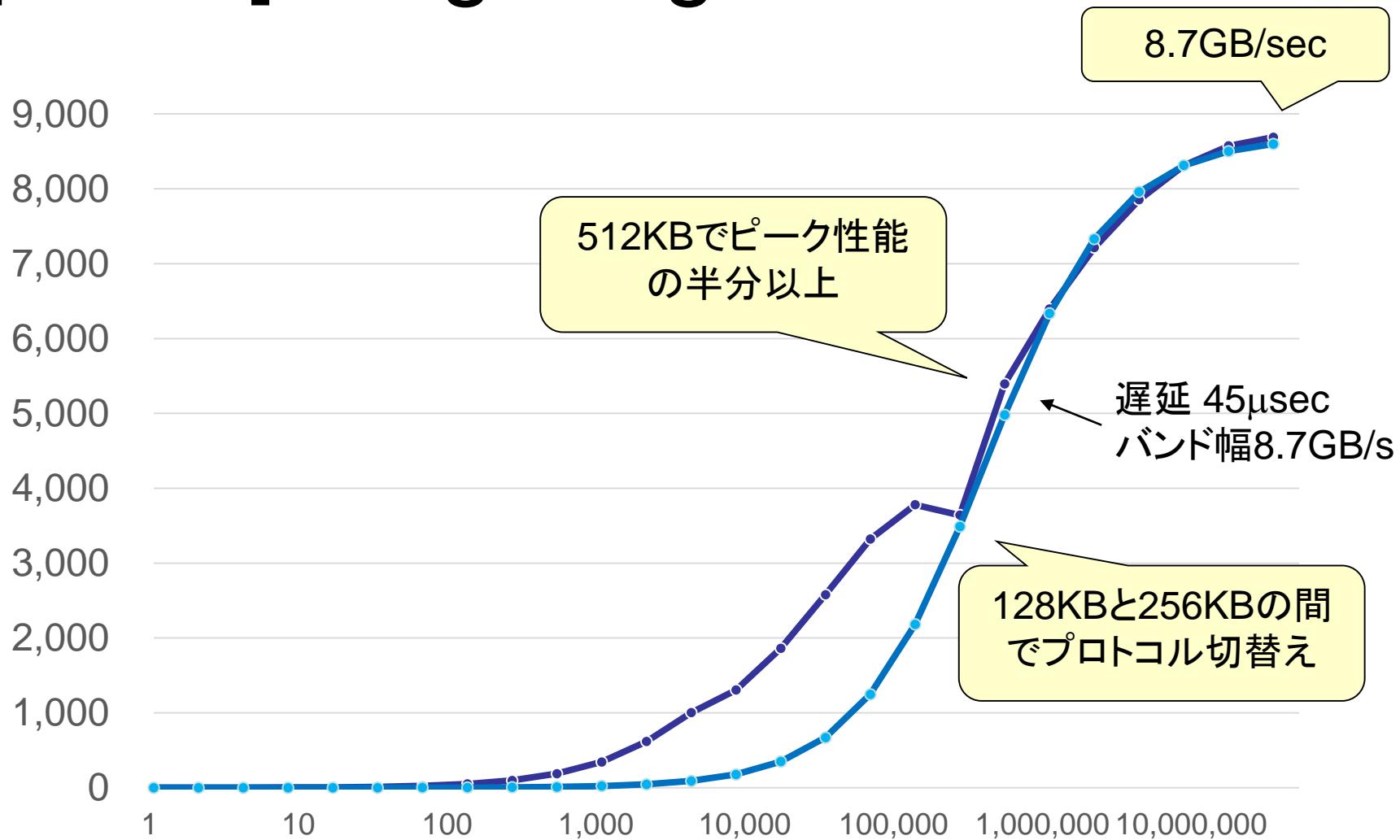
$L$  遅延時間       $B$  バンド幅

# [環境1] PingPongベンチマークの 考察



- データサイズは大きい方が高性能
- 参考：理論ピーク性能は113.1MB/sec
- ピークの半分以上→データサイズ16KB以上
- ピークの9割以上→データサイズ512KB以上
  
- 1バイトのPingPongベンチマークの遅延時間は563 $\mu$ secであったが、ロングメッセージは200 $\mu$ secの遅延時間の曲線に従う

# [環境2] PingPongベンチマーク



# [環境2] PingPongベンチマークの 考察



- ピークの半分以上→データサイズ512KB以上
- ショートメッセージの遅延は2 $\mu$ 秒であるが、ロングメッセージは45 $\mu$ secの遅延時間の曲線に従う





# Intel® MPI Benchmark

- 基本MPIベンチマークカーネル
- MPI1

– PingPong	<b>Single</b>
– PingPing	<b>Transfer</b>
– Sendrecv	<b>Parallel</b>
– Exchange*	<b>Transfer</b>
– Bcast	<b>Collective</b>
– Allgather	
– Allgatherv	
– Alltoall*	
– Alltoallv*	
– Reduce	
– Reduce_scatter	
– Allreduce*	
– Barrier	
– 上記を複数一斉に行うMulti版	

- EXT
  - Window
  - Unidir\_Put
  - Unidir\_Get
  - Bidir\_Get
  - Bidir\_Put
  - Accumulate
- IO
  - S\_{Write,Read}\_{indv,expl}
  - P\_{Write,Read}\_{indv,expl,shared,priv}
  - C\_{Write,Read}\_{indv,expl,shared}
- NBC – nonblocking collective
- RMA – MPI3 RMA
- MT – multithreaded MPI1
- P2P



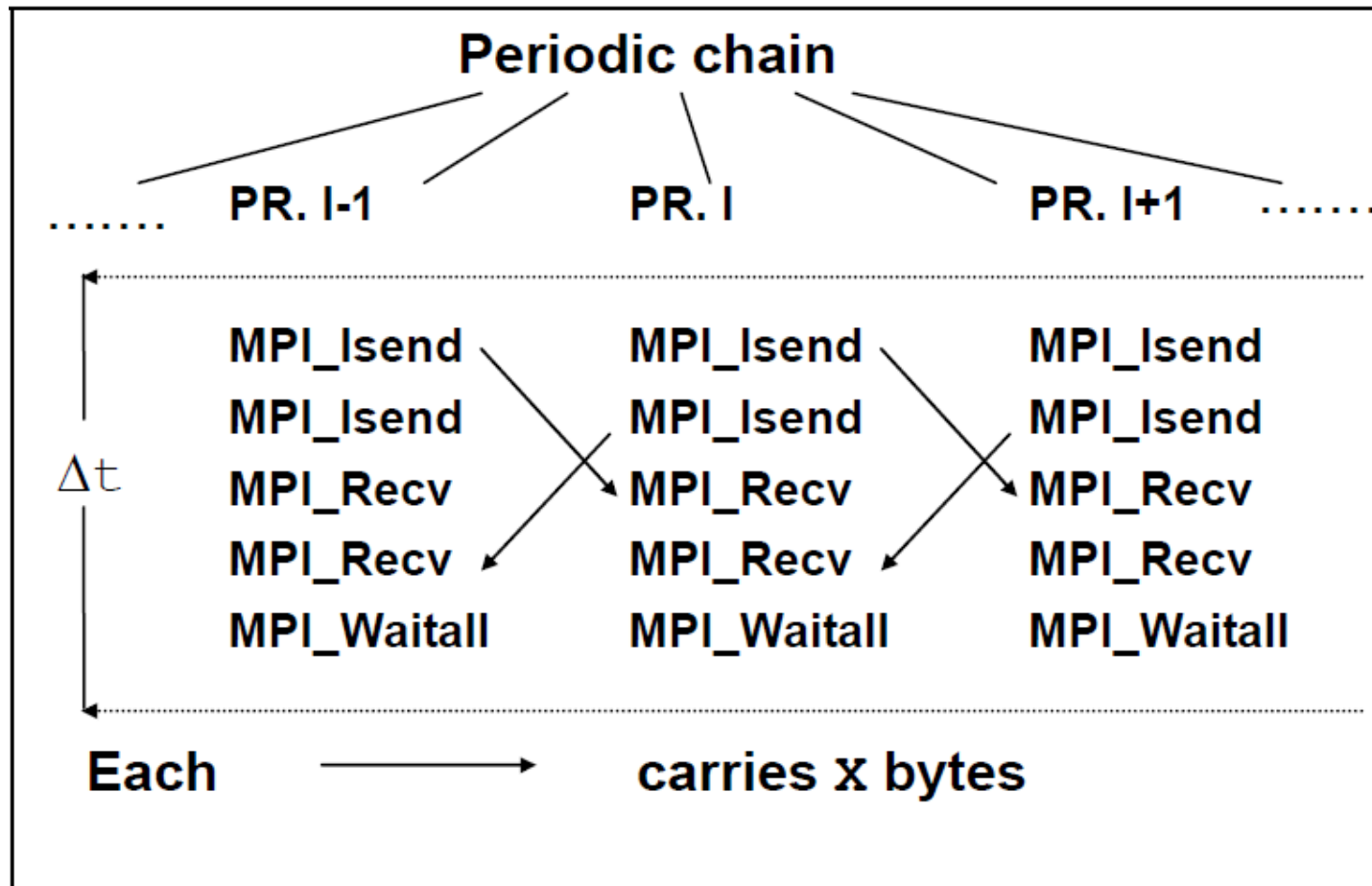
# IMB – How to install

```
% git clone https://github.com/intel/mpi-benchmarks IMB  
% cd IMB  
% module load openmpi  
% CC=mpicc CXX=mpicxx make
```



# Exchangeパターン

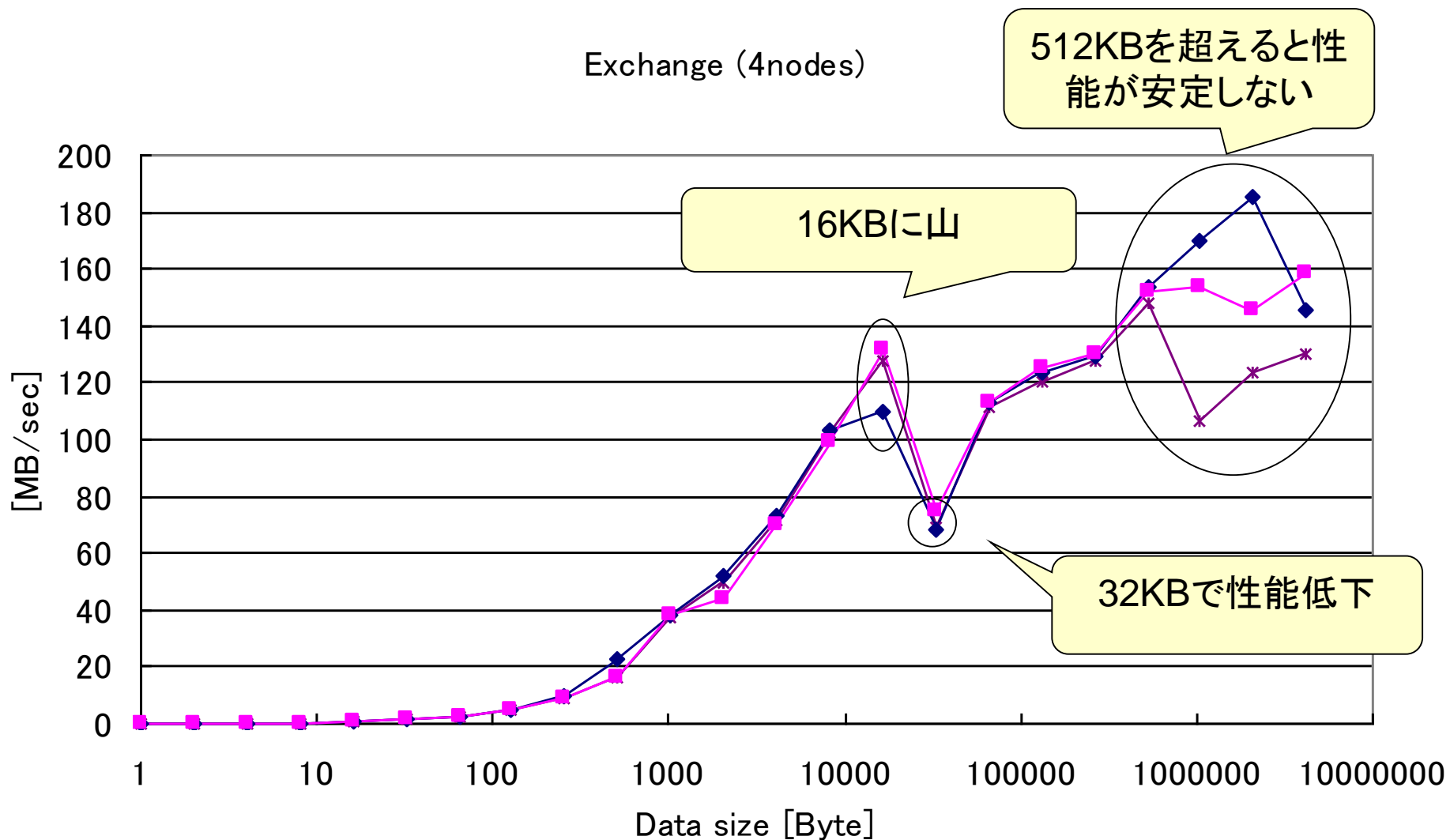
- 境界の要素を交換する通信パターン



\*Intel MPI Benchmarks Users Guide and Methodology Descriptionより



# [環境1] Exchange (4ノード) [試行3回]



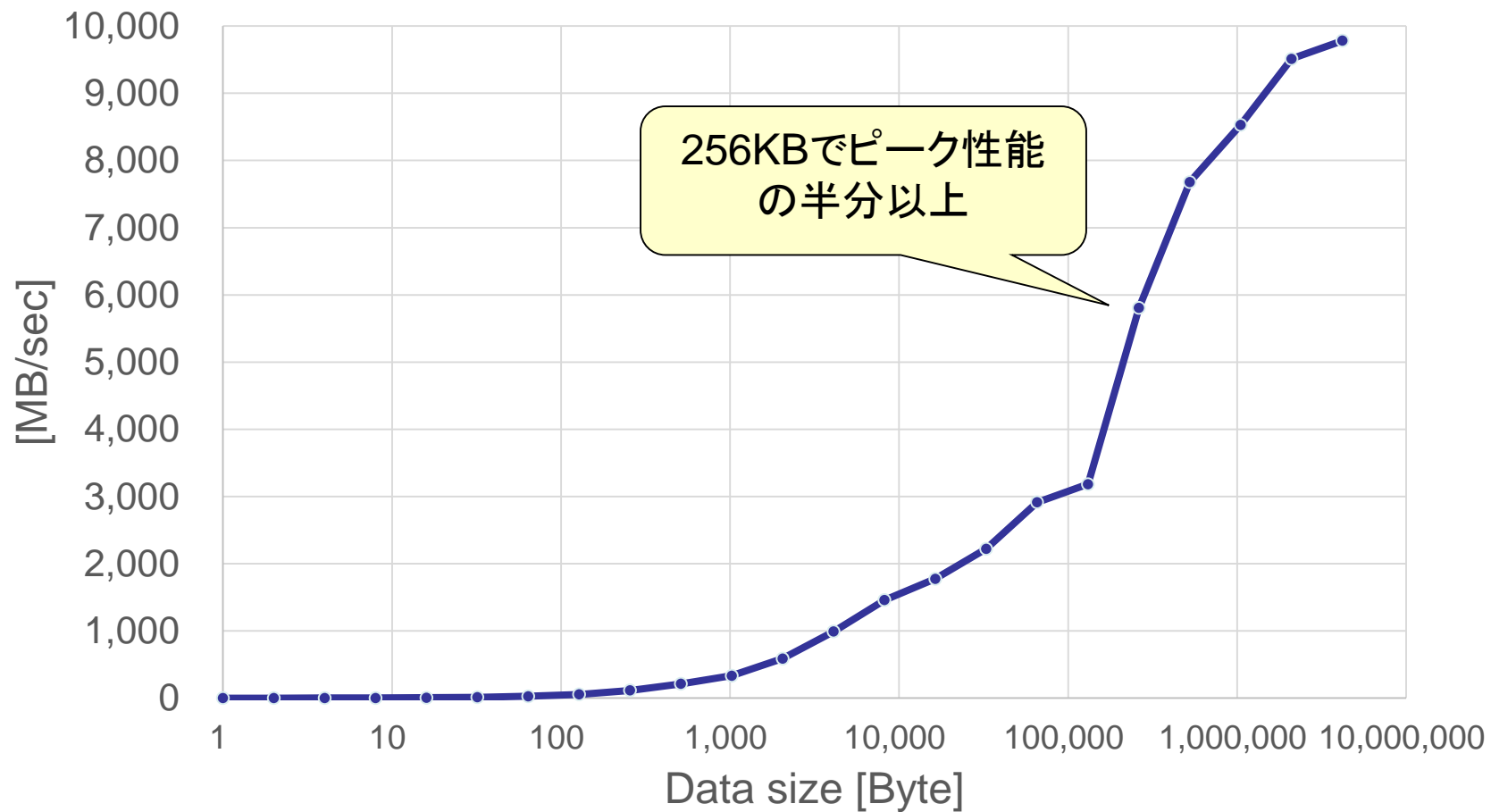


# [環境1] Exchange (4ノード) の考察

- データサイズは基本的には大きい方が高性能だが、32KB付近で落ちる
- 参考：理論ピーク性能は  
 $2 * 113.1 = 226.2 \text{ MB/sec}$
- ピークの半分以上 → データサイズ16KBと128KB以上
  - 32KB、64KBはピークの半分以下
- 512KB超では性能が安定しない
  - パケットロスとRTOのため



# [環境2] Exchange (4ノード)





## [環境2] Exchangeの考察

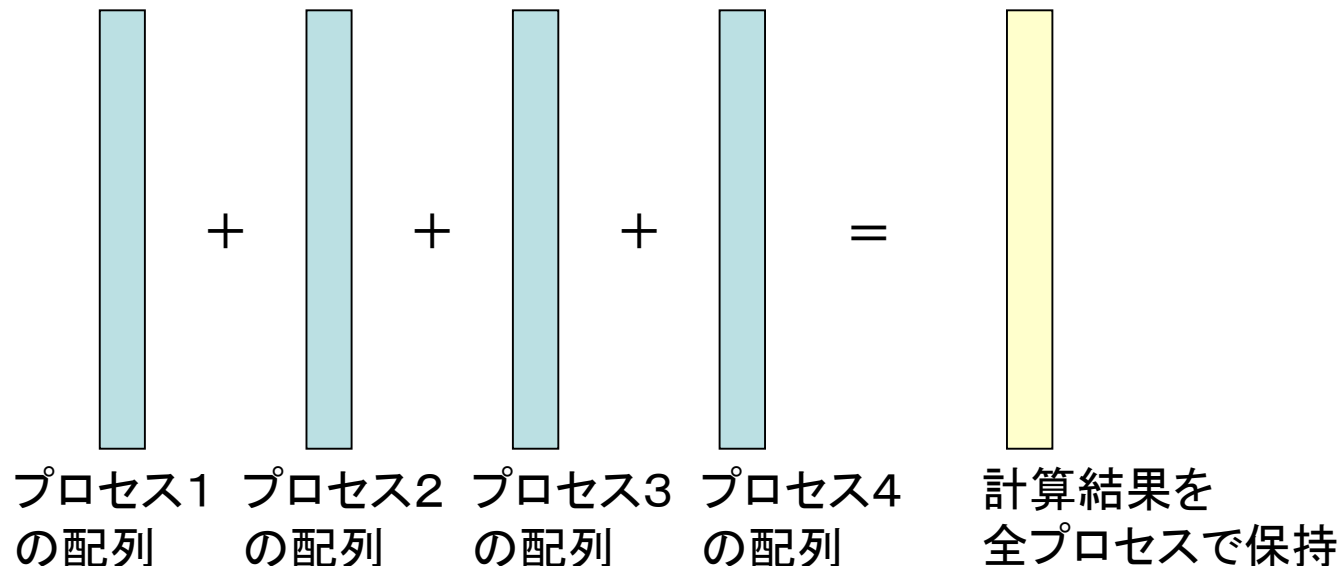
- データサイズは基本的には大きい方が高性能
- 256KB以上でピークの半分以上
- 性能は安定している(パケットが落ちない)



# Allreduce

- 各プロセスの配列間で指定された演算（加算、AND/OR演算など）を施し結果は全プロセスで保持
- MPI\_SUMの例

$$x_1 + x_2 + x_3 + x_4 = \sum_{i=1}^4 x_i$$

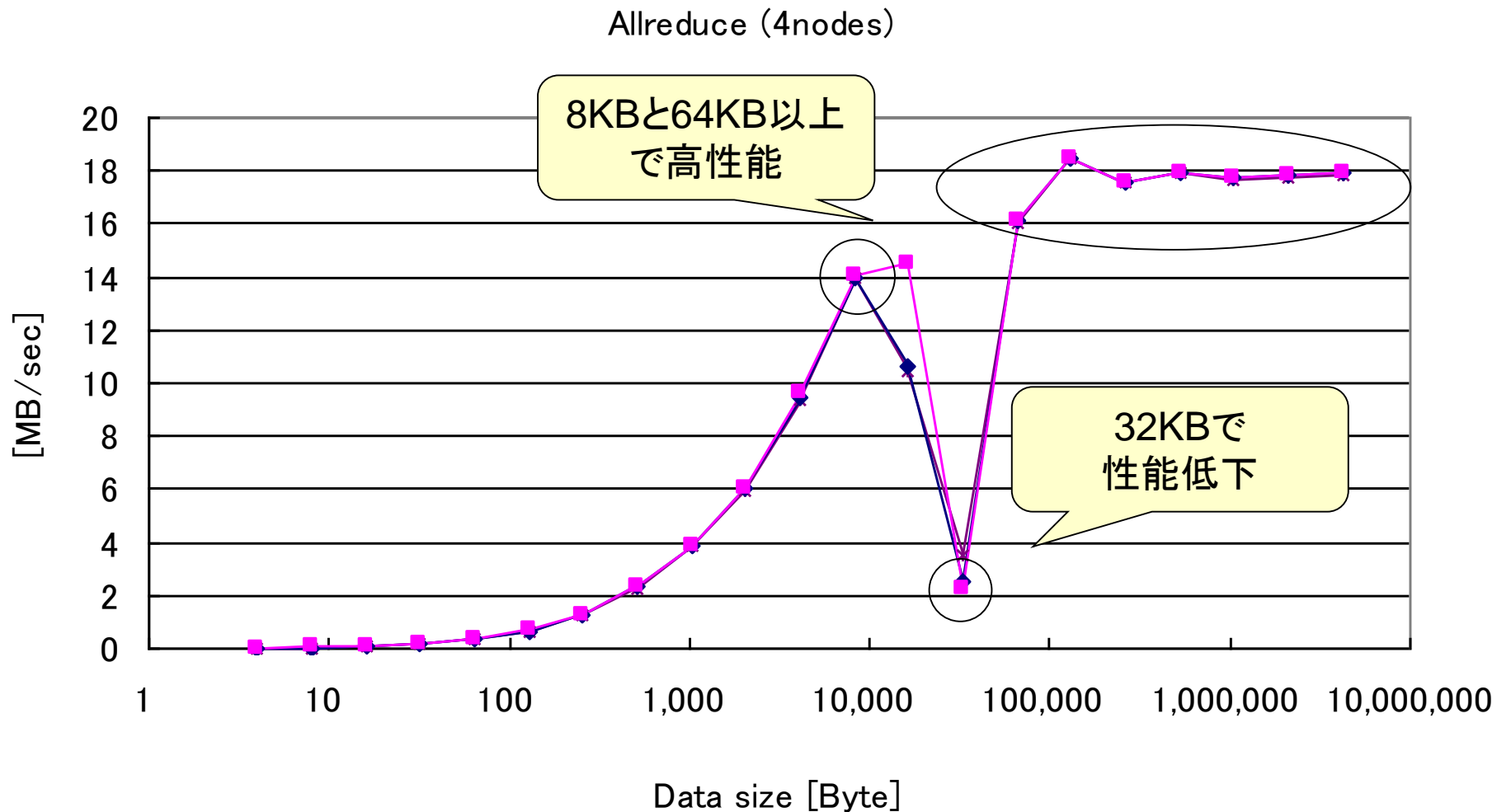






# [環境1] Allreduce (4ノード)

## [サイズ/時間]



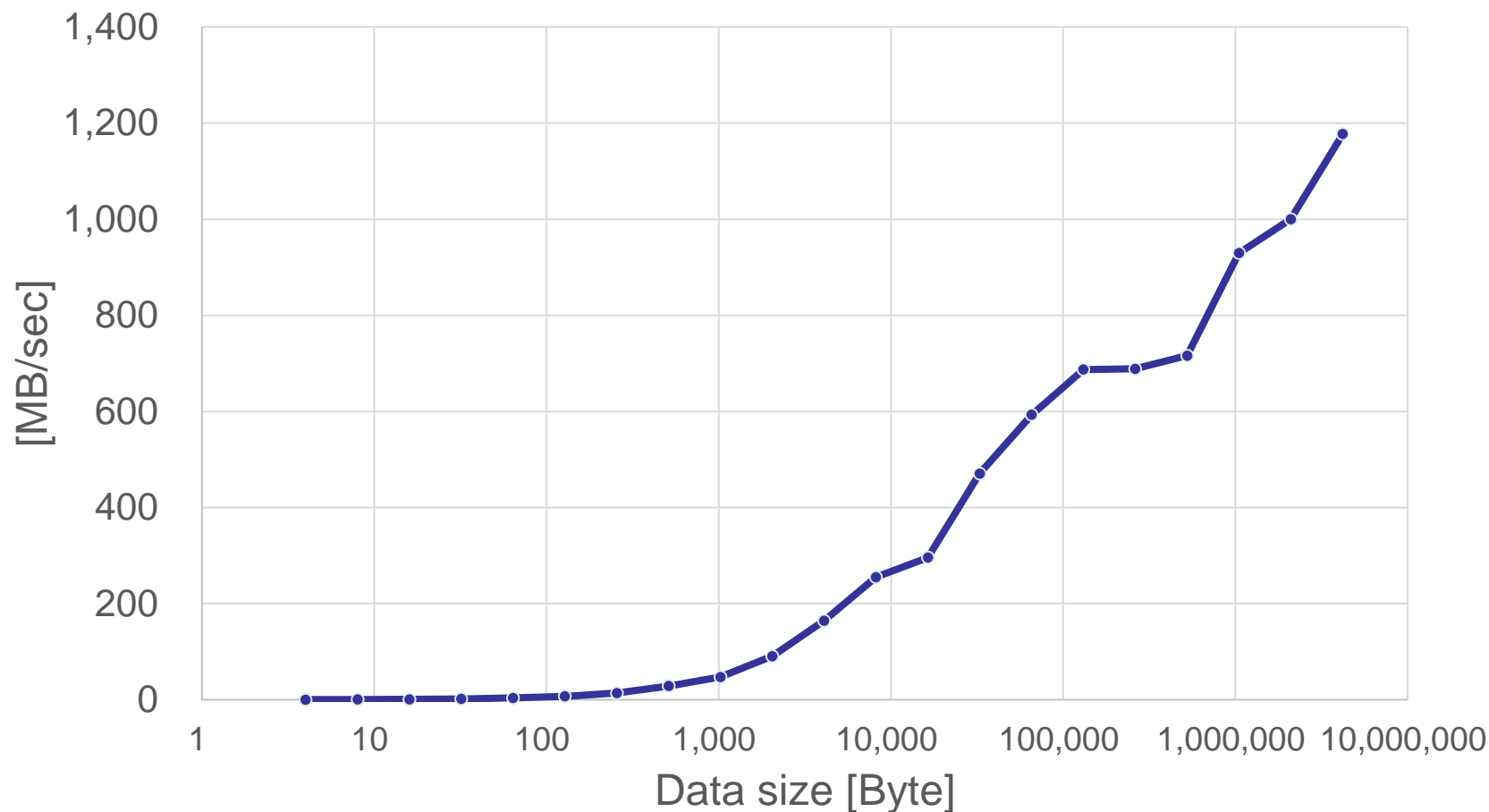


# [環境1] Allreduceの考察

- データサイズは基本的には大きい方が高性能だが、32KBで性能低下
- 8KB、64KB以上では高性能

# [環境2] Allreduce (4ノード)

## [サイズ/時間]





## [環境2] Allreduceの考察

- データサイズは基本的には大きい方が高性能
- 性能は安定している



# プロファイラ

- プログラムの挙動を把握する
  - 呼び出し回数の多い関数
  - 処理に時間がかかっている関数
  - 関数の呼び出し関係
  - 関数のメモリ使用量など
- 実行時間の多くが費やされているコードの特定
- 並列プログラムにおける同期待ち、負荷不均衡の把握
  - プログラムの実行に影響しないことが望ましい
  - 軽量のプロファイラが必須



# 計時コード挿入によるプロファイリング

- 計時したい箇所 (MPI関数、特定ブロック) に計時コードを挿入

```
double t;
```

```
t = MPI_Wtime();
```

```
MPI_Allgather(...);
```

```
t = MPI_Wtime() - t;
```

- 時間精度はシステム依存



# tlog – time log

- 実行プロファイルをとるための軽量ライブラリ
  - 1イベントあたり16バイト
  - 各プロセスのメモリに保持
- 単発イベント、区間イベント各9種類のログ
  - イベント番号は8ビットなので拡張可能
- tlog\_initializeからの経過時間(秒)を記録
  - tlog\_initializeでノード間の時刻差を測定し補正
  - 並列プロセスにおける「絶対」相対時間
- 暫定ダウンロードURL
  - <http://www2.ccs.tsukuba.ac.jp/workshop/HPCseminar/2011/software/tlog-0.9.tar.gz>



# tlog - 主要API

void tlog\_initialize(void)

初期化。MPI\_Initの後で呼ぶこと

void tlog\_log(int event)

eventで指定されたイベントを記録する

void tlog\_finalize(void)

ログをtrace.logに出力。MPI\_Finalize()の前に呼ぶこと

```
tlog_initialize();  
...  
tlog_log(TLOG_EVENT_1_IN);  
/* EVENT 1 */  
tlog_log(TLOG_EVENT_1_OUT);  
...  
tlog_finalize();
```





# 例 - cpi.c

- $\pi$ を計算するテストプログラム

```
MPI_Init(&argc, &argv);
tlog_initialize();
tlog_log(TLOG_EVENT_1_IN);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
tlog_log(TLOG_EVENT_1_OUT);
/* mypiの部分計算 */
tlog_log(TLOG_EVENT_2_IN);
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
tlog_log(TLOG_EVENT_2_OUT);
if (rank == 0) /* 結果表示 */
tlog_log(TLOG_EVENT_1_IN);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
tlog_log(TLOG_EVENT_1_OUT);
tlog_finalize();
MPI_Finalize();
```



# 例 - cpiのコンパイル

- tlogライブラリをリンク

```
% mpicc -O -o cpi cpi.c -ltlog
```

- tlogライブラリ, tlogviewのインストール

```
% ./configure
```

```
% make
```

```
% sudo make install
```

/usr/localにインストール  
する例



# 例 - cpiの実行結果

```
$ mpiexec -hostfile hosts -n 4 cpi
adjust i=1,t1=0.011781,t2=0.011886,t0=0.011769,diff=6.7e-05
adjust i=2,t1=0.012911,t2=0.013015,t0=0.012877,diff=8.8e-05
adjust i=3,t1=0.014441,t2=0.014548,t0=0.014392,diff=0.000115
adjust i=1,t1=0.01623,t2=0.016335,t0=0.016285,diff=-2e-06
adjust i=2,t1=0.017314,t2=0.017418,t0=0.017367,diff=-2e-06
adjust i=3,t1=0.018401,t2=0.018504,t0=0.018454,diff=2.5e-06
tlog on ...
Process 0 on exp0.omni.hpcc.jp
pi is approximately 3.1416009869231249, Error is 0.00000833333333318
wall clock time = 0.000213
tlog finalizing ...
Process 3 on exp3.omni.hpcc.jp
Process 1 on exp1.omni.hpcc.jp
Process 2 on exp2.omni.hpcc.jp
tlog dump done ...
```

ノード間の  
時間差測定  
(デバッグ時に  
出力)

デバッグ時の出力

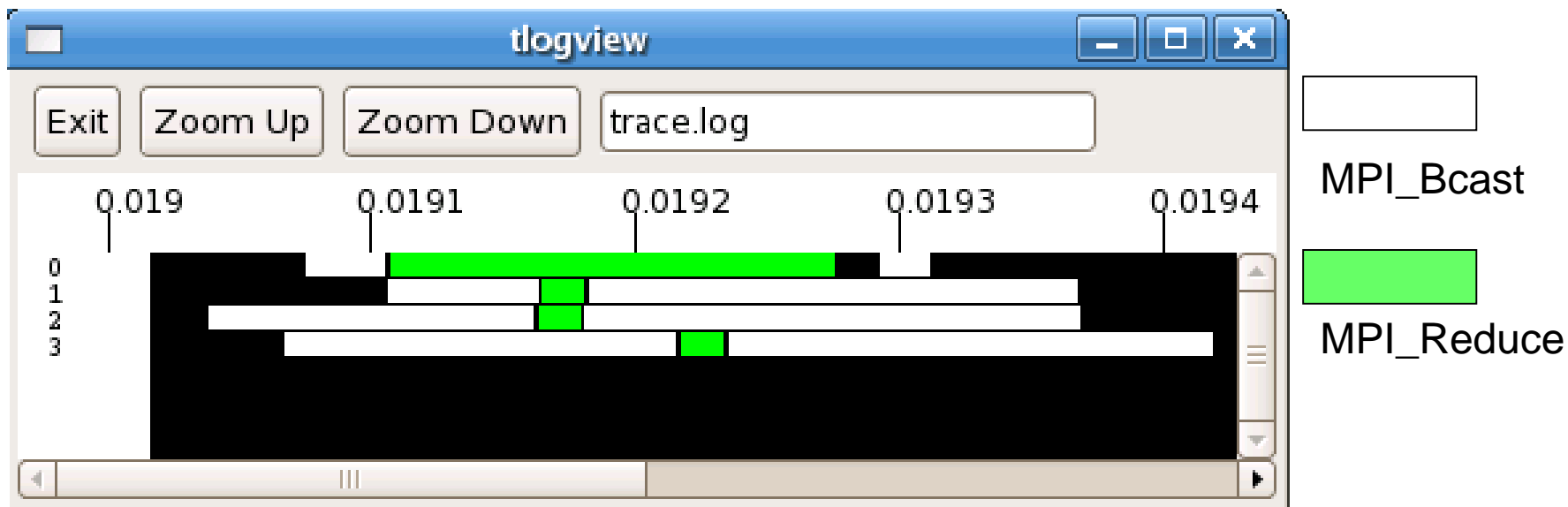
プログラムの  
出力

デバッグ時の出力



# cpiのプロファイル結果(1)

- tlogview – tlogの可視化ツール  
% tlogview trace.log
- 4プロセス(4ノード)での実行プロファイル

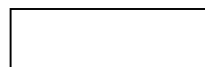
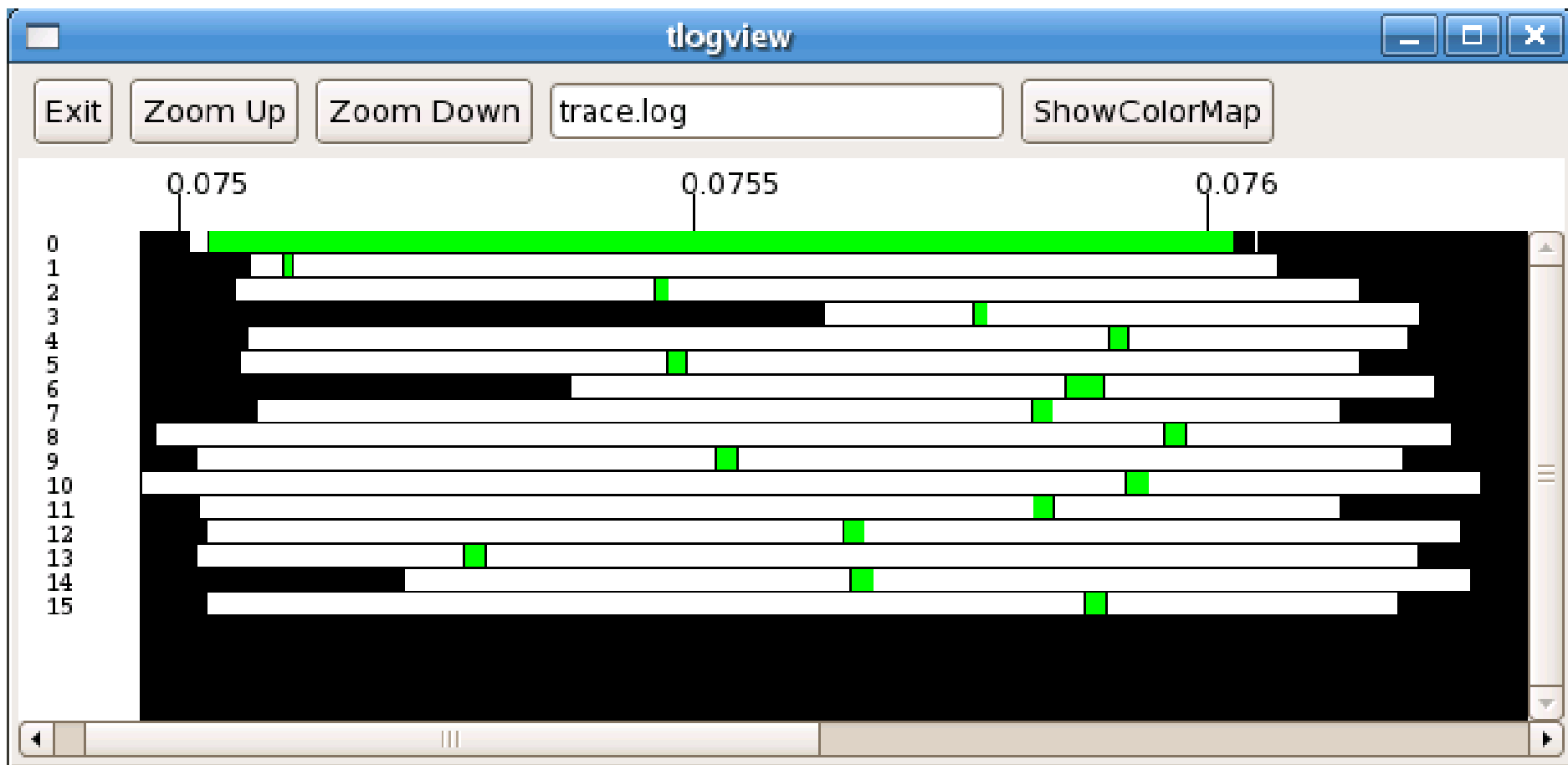


tlog\_initializeからの経過時間(秒)。ノード間の時刻差修正済。



# cpiのプロファイル結果(2)

- 16プロセス(4ノード×4プロセス)のプロファイル



MPI\_Bcast



MPI\_Reduce



# 通信最適化

- 通信の削減
- 負荷分散\*
- 基本的には通信データサイズを大きく
  - 通信ブロック
  - 複数反復をまとめる
- 通信遅延隠蔽
  - 通信と計算のオーバラップ
  - パイプライン実行



# 通信の削減

```
MPI_Reduce(&xx, &x, 1, MPI_DOUBLE,  
            MPI_SUM, 0, MPI_COMM_WORLD);
```

```
MPI_Reduce(&yy, &y, 1, MPI_DOUBLE,  
            MPI_SUM, 0, MPI_COMM_WORLD);
```

```
MPI_Reduce(&zz, &z, 1, MPI_DOUBLE,  
            MPI_SUM, 0, MPI_COMM_WORLD);
```



```
MPI_Reduce(xx, x, 3, MPI_DOUBLE,  
            MPI_SUM, 0, MPI_COMM_WORLD);
```



# 負荷分散

- MPIはSPMDで、集団通信などで同期がとられる
- 一番遅いプロセスを待つこととなる
- 計算時間を均一にすることが大切





# 通信ブロッキング

- 1対1通信、集団通信はデータサイズによって通信性能が大きく変化する
- 通信ブロッキングは、通信データをまとめてデータサイズを変更する(大きくする)手法
  - データのブロック分散
  - 複数反復をまとめる(テンポラルブロッキング)



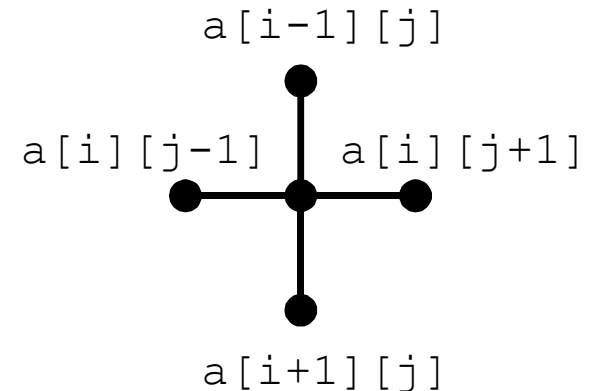
# 通信ブロッキングの例：ヤコビ法

- 二次元ポアソン方程式を5点差分で離散化した連立一次方程式の解法

```

jacobi() {
  while (!converge) {
    for(i = 1; i < N - 1; ++i)
      for(j = 1; j < N - 1; ++j)
        b[i][j] = .25 *
          (a[i - 1][j] + a[i][j - 1]
           + a[i][j + 1] + a[i + 1][j]);
    /* 収束テスト */
    /* bをaにコピー */
  }
}

```

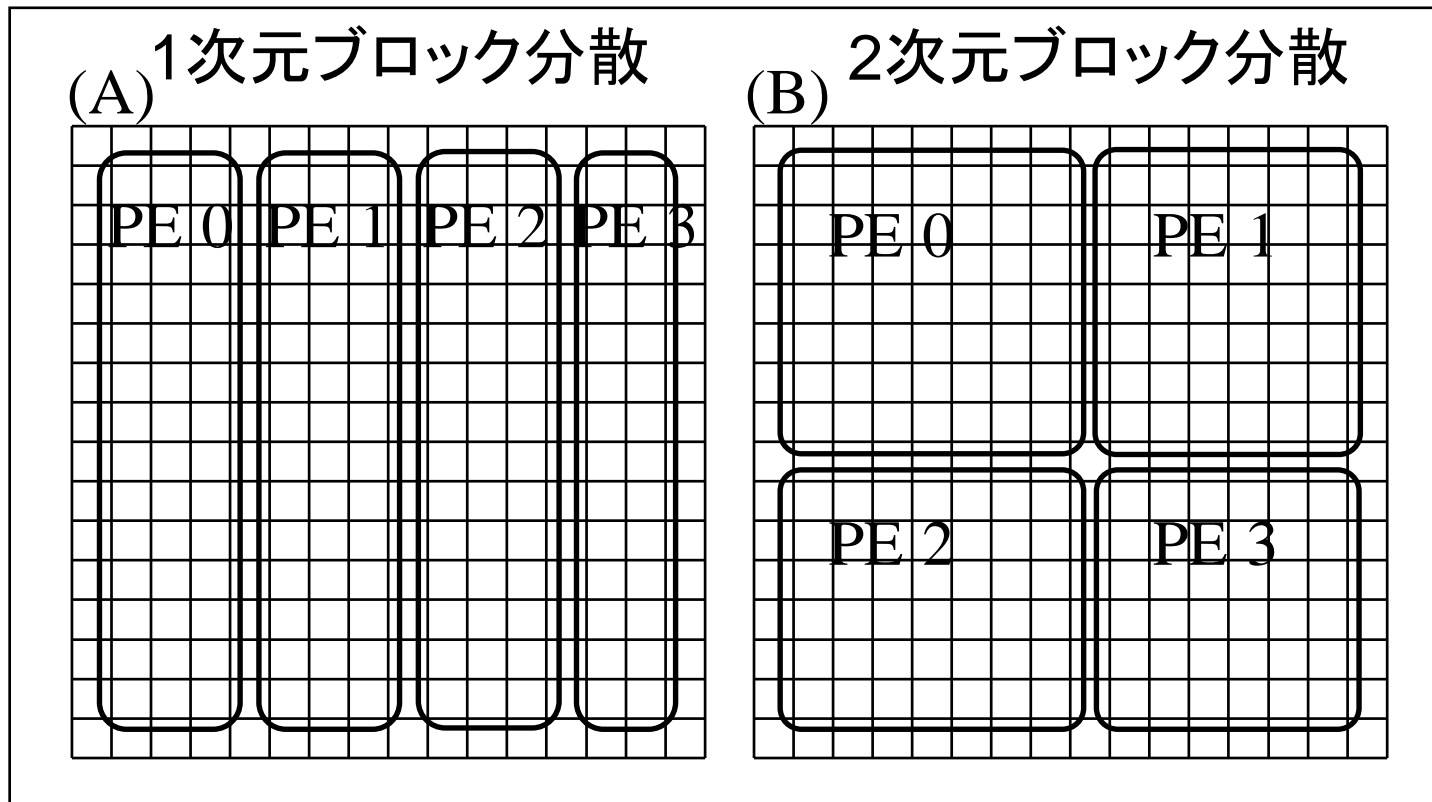


データ依存関係

\*本当はヤコビ法ではなくRB-SOR法などを使って欲しい








# データのブロック分散

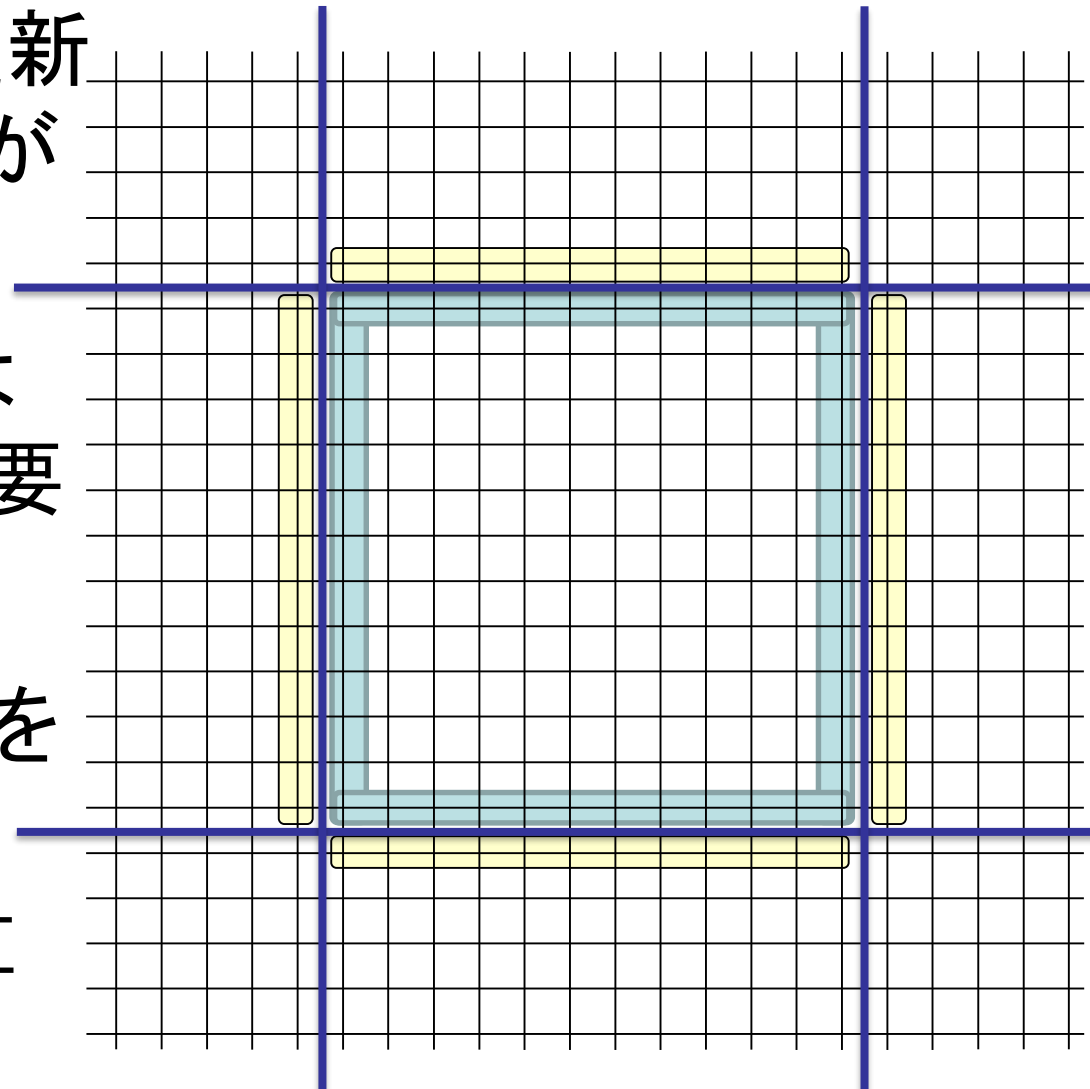


- データをブロック分割することにより通信データサイズを大きくできる
  - 1次元ブロック分散では  $n$
  - 2次元ブロック分散では  $n/\sqrt{p}$







# シャドウ領域（袖領域）の通信

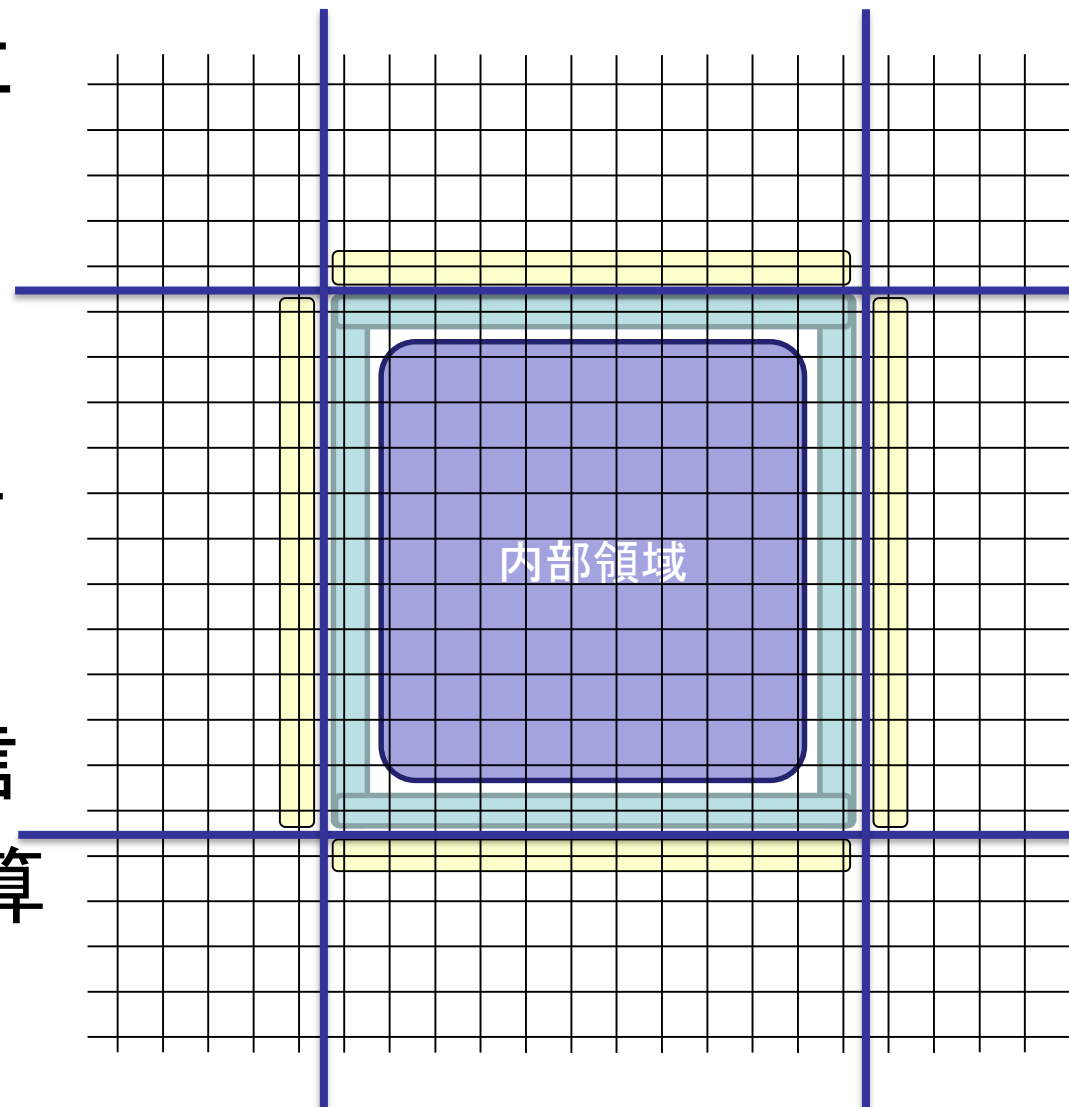
- 境界領域  の更新には  のデータが必要
  - 他のプロセスでは  のデータが必要
1.  と  のデータを一括して交換
  2. 各プロセスで計算





# 計算と通信のオーバーラップ

- 内部領域の更新には  のデータは不要
1.  のデータを送信
  2. 内部領域の計算
  3.  のデータの受信
  4. 境界領域  の計算

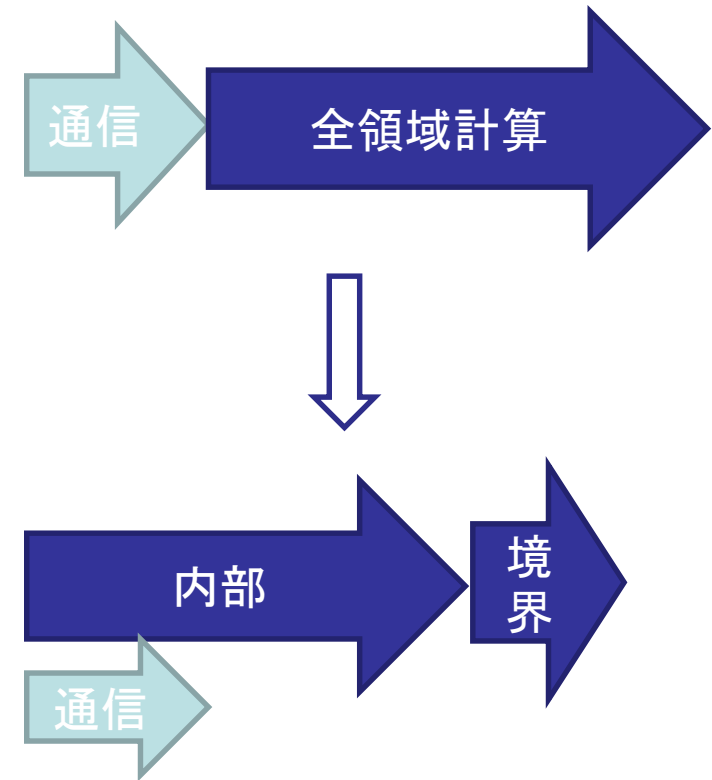




# 計算と通信のオーバラップ(2)

- MPI\_Isend( , ..., &req[0])
- MPI\_Irecv( , ..., &req[1])
- 内部領域計算
- MPI\_Waitall(2, req, status)
- 境界領域計算

内部領域の計算と通信を  
オーバラップさせて通信遅延隠蔽





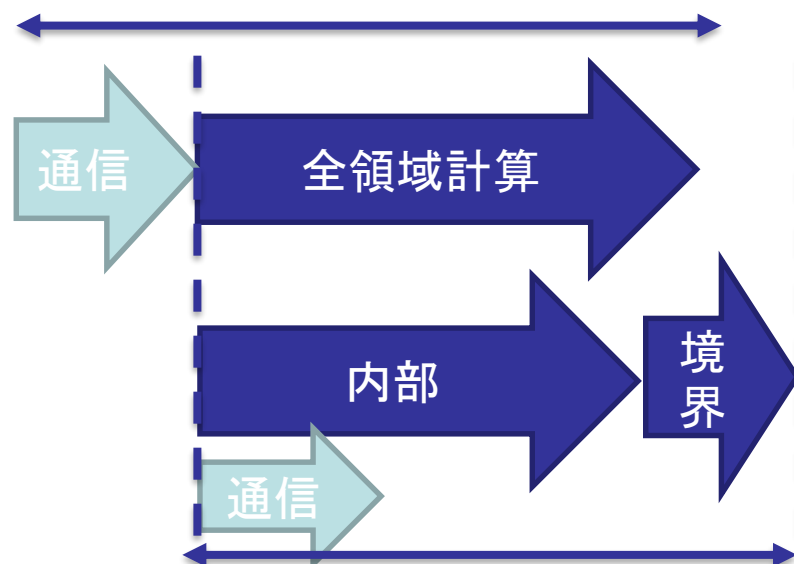
# 計算と通信のオーバーラップについての注意

- 性能劣化の原因にもなる

- 境界領域の計算はキャッシュミスを起こしやすく、演算性能が低下

- 全領域計算 < 内部領域計算 + 境界領域計算

- 内部 + 境界 < 全領域 + 通信でなければ性能劣化



演算時間は増大



# 複数反復をまとめる(テンポラルブ ロッキング)

- ヤコビ法二反復をまとめる

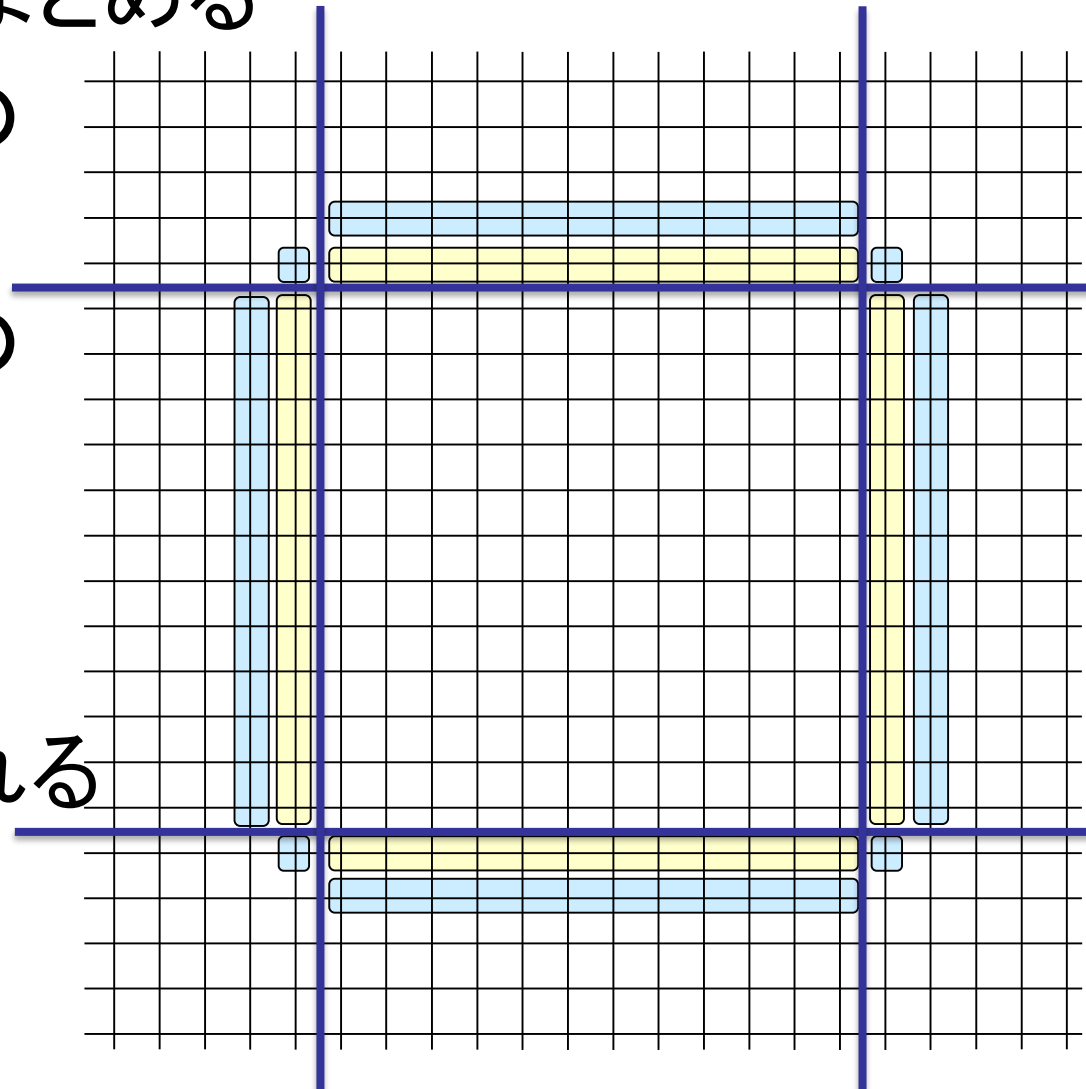
- 一反復目では  のデータが必要

- 二反復目では  のデータが必要

- と  のデータを転送することにより二反復をまとめられる

– 1次元  $2n$

– 2次元  $2n / \sqrt{p}$

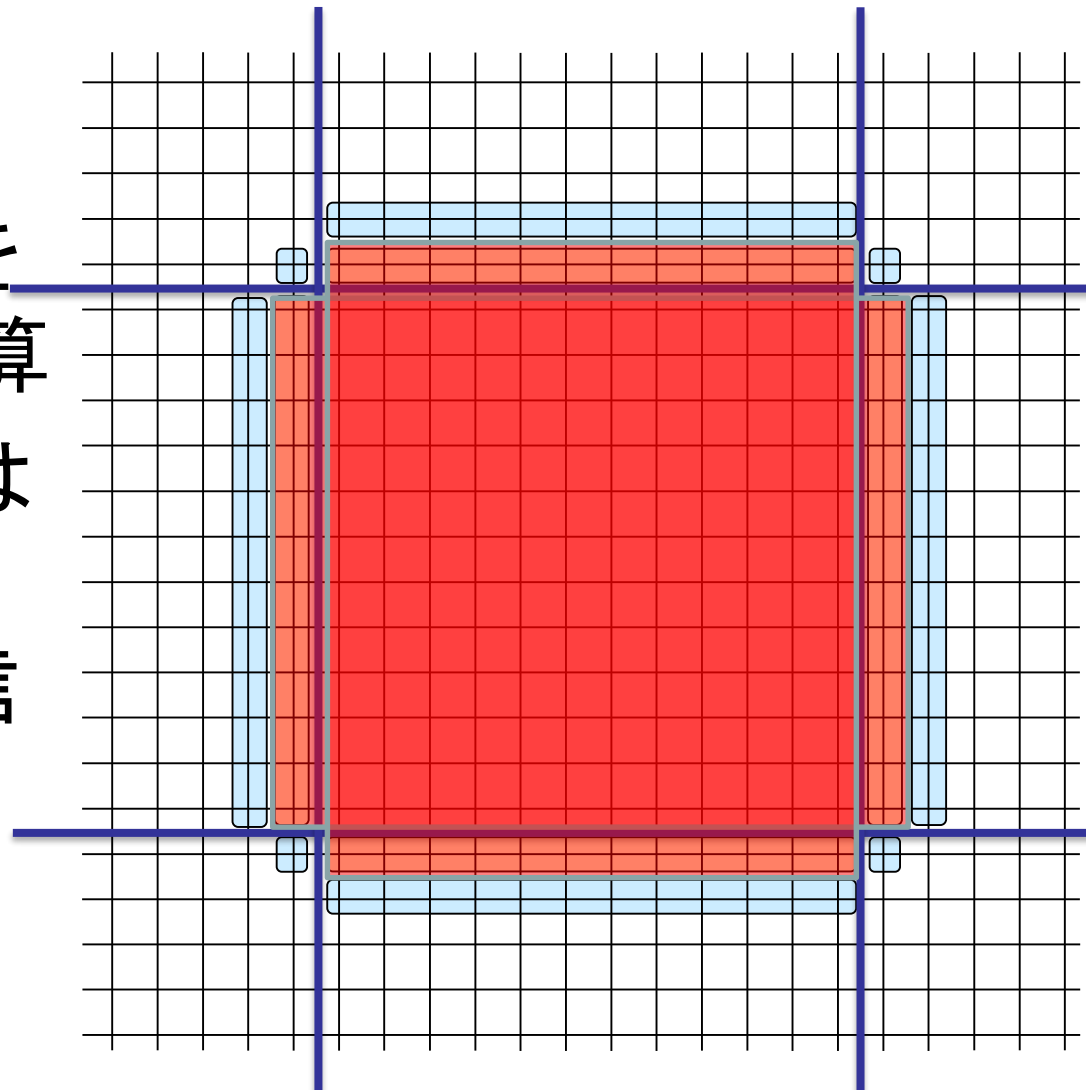






## 複数反復をまとめる(2)

- □ と □ のデータを転送する
- [一反復目]袖領域を含めた赤領域を計算
- [二反復目]袖領域は既に最新データを持っているため通信なしに計算可能





# まとめ

- 基本通信性能
  - 1対1通信
  - 集団通信
- プロファイラ
- 通信最適化
  - 通信の削減
  - 通信遅延隠蔽
  - 通信ブロック
  - 負荷分散



# 「最適化2」レポート課題

- 2次元ラプラス方程式をヤコビ法で解くMPIプログラム(参考:「MPI」で紹介したlaplace)を作成し, 複数反復をまとめる最適化を行いなさい。最適化前後でtlogによるプロファイルを行い, 考察を行いなさい。