



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

建部修見

tatebe@cs.tsukuba.ac.jp

筑波大学大学院システム情報工学研究科
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



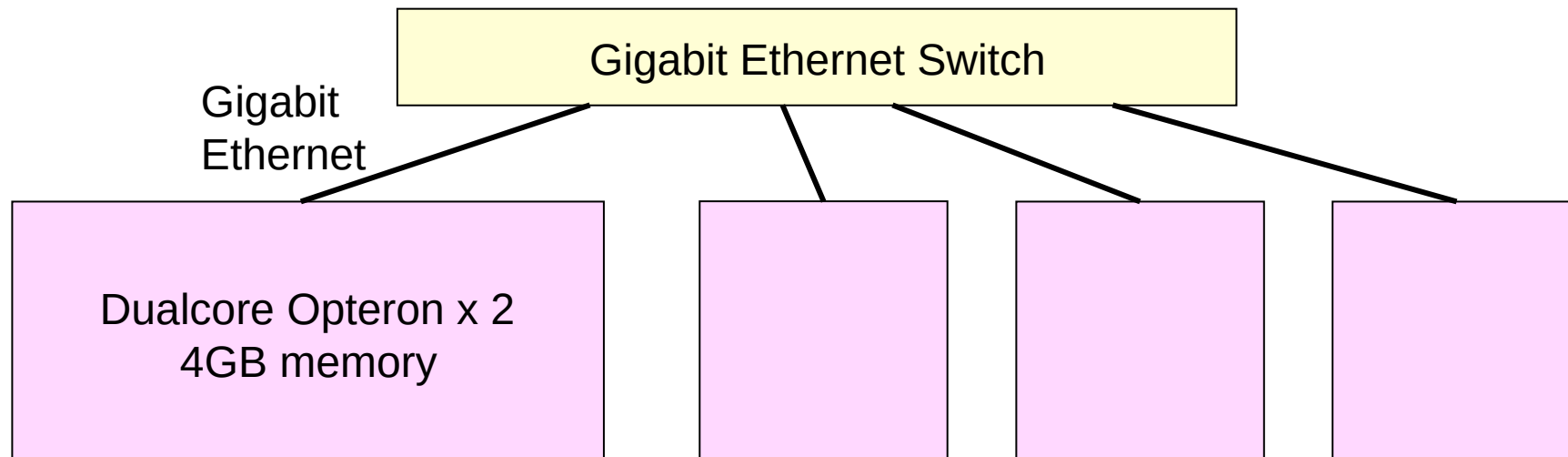
基本通信性能

- 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)

- T2K筑波4ノード
 - 2.3GHz Quadcore Opteron x 4 sockets (16 cores)
 - 32GB memory
 - MVAPICH2
- 4xDDR Infinibandで接続
 - 理論ピーク性能は8GB/sec
- メモリ配置などの最適化は施さず



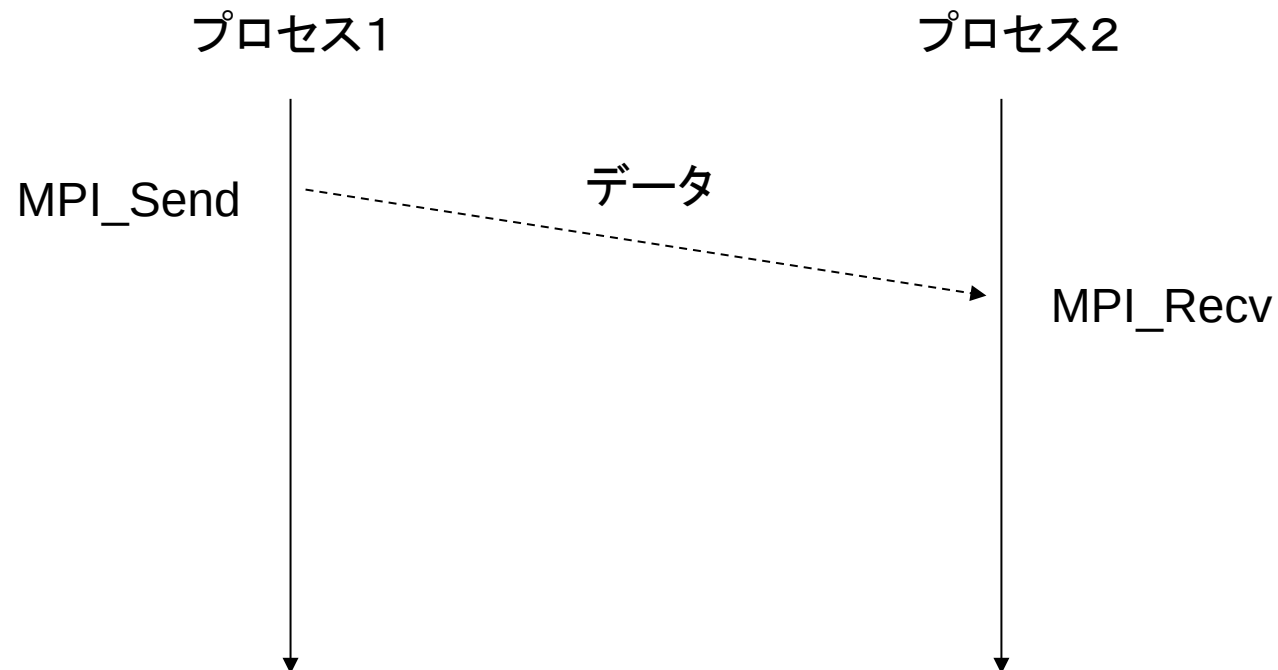
基本通信性能評価環境(3)

- T2K京都4ノード
 - 2.3GHz Quadcore Opteron x 4 sockets (16 cores)
 - 32GB memory
 - Fujitsu Parallelnavi MPI
- 4xDDR Infinibandで接続
 - 理論ピーク性能は8GB/sec
- メモリ配置などの最適化は施さず



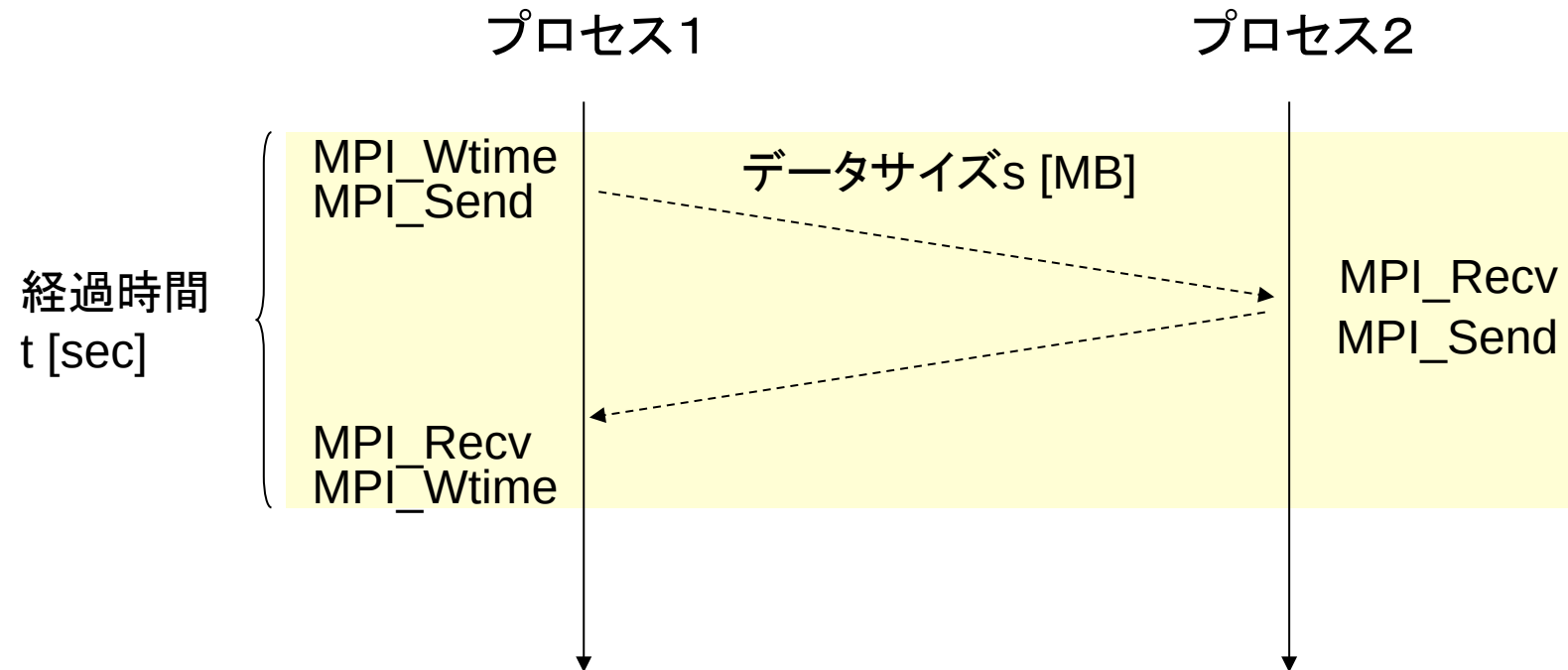
1対1通信の性能

- 1対1通信はMPIにおける基本通信





PingPongベンチマーク



通信バンド幅 $s/t/2$ [MByte/sec]

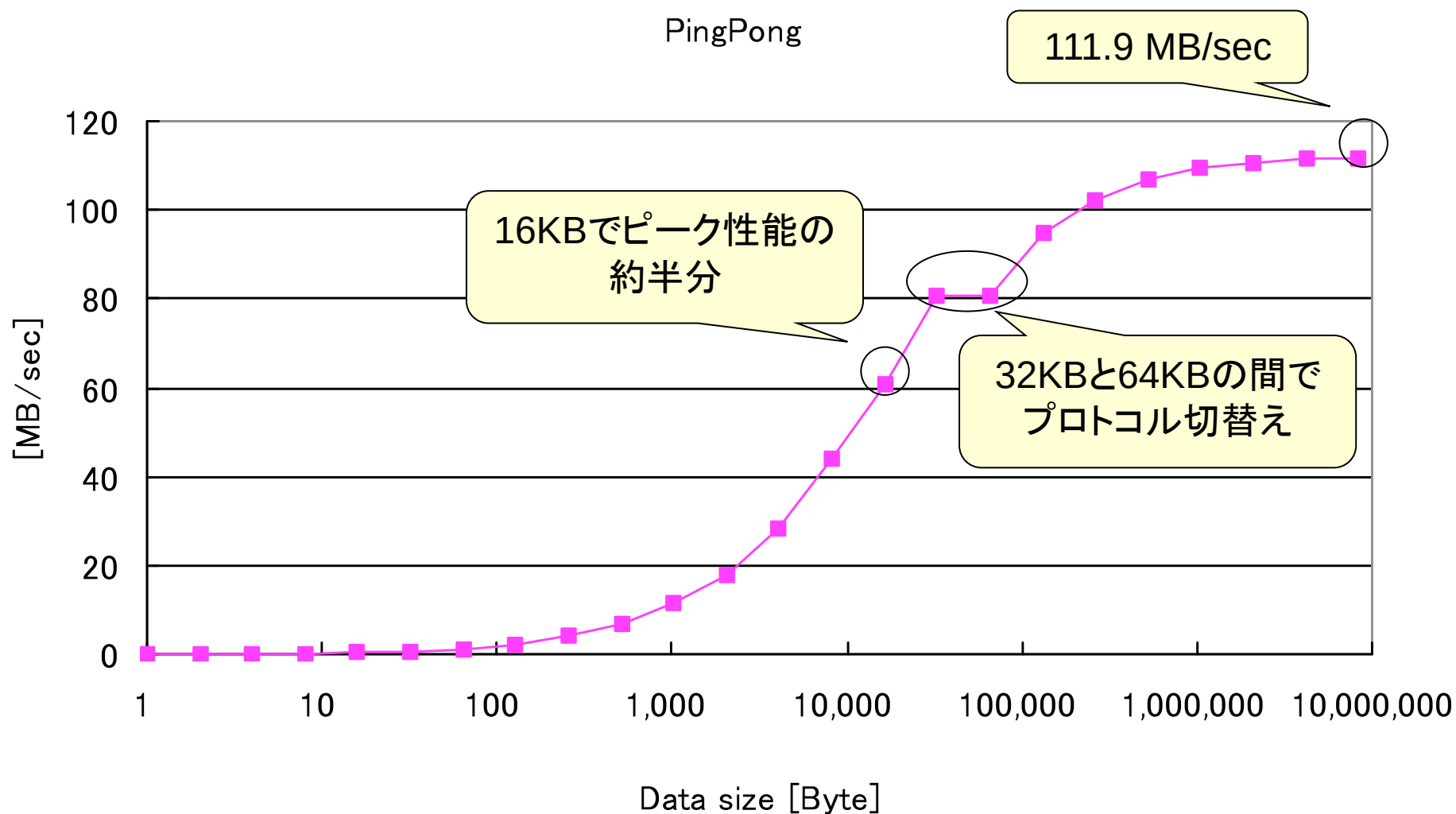


PingPongベンチマークの例

```
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  
}
```

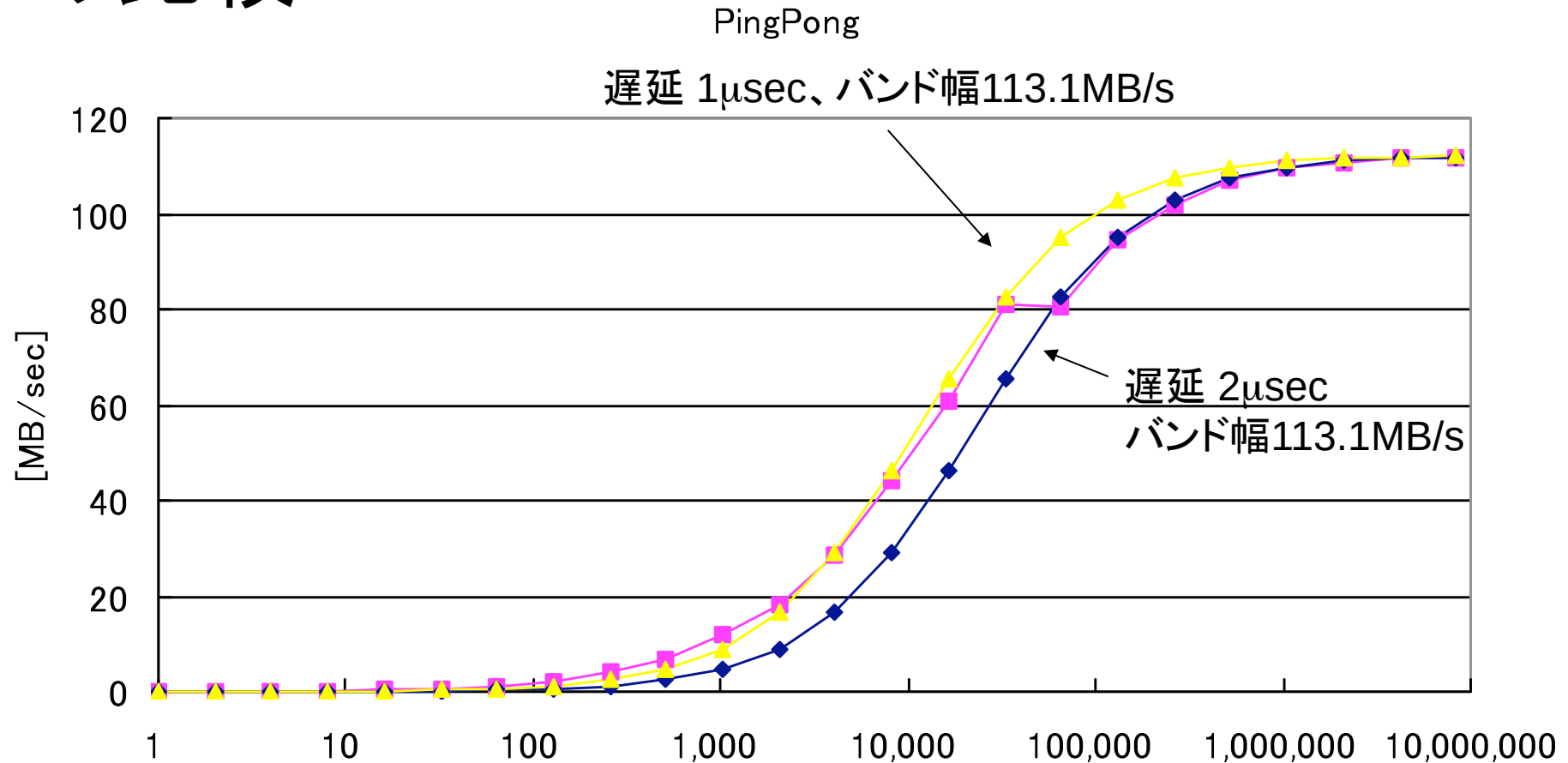


[環境1] PingPongベンチマーク





[環境1] 遅延、バンド幅での曲線と の比較



理論曲線 $s / (L + s / B)$

Data size [Byte]

L 遅延時間

B バンド幅



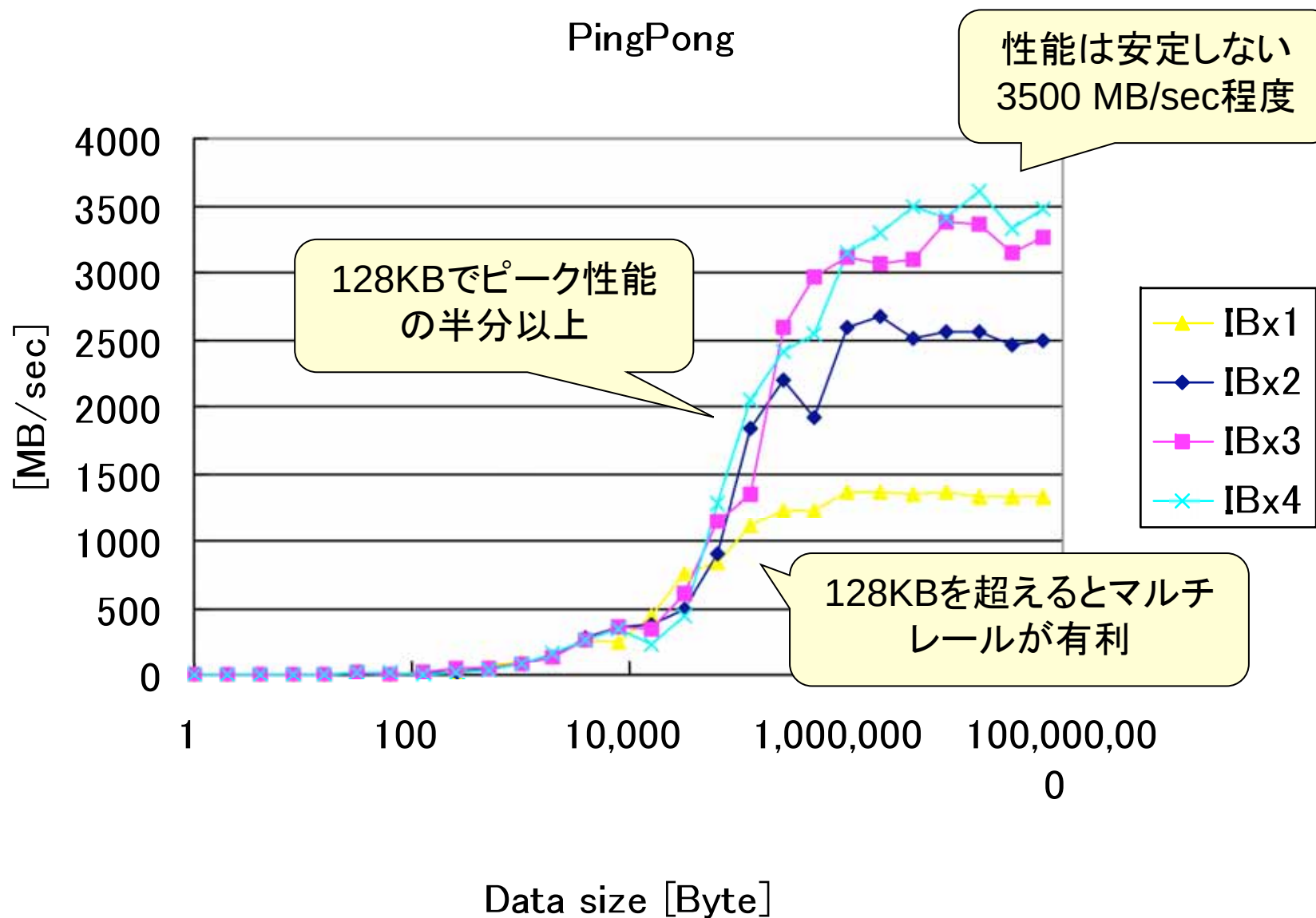
[環境1] PingPongベンチマークの考察

- 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52msecであったが、ショートメッセージは1msec、ロングメッセージは2msecの遅延時間の曲線に従う



[環境2] PingPongベンチマーク

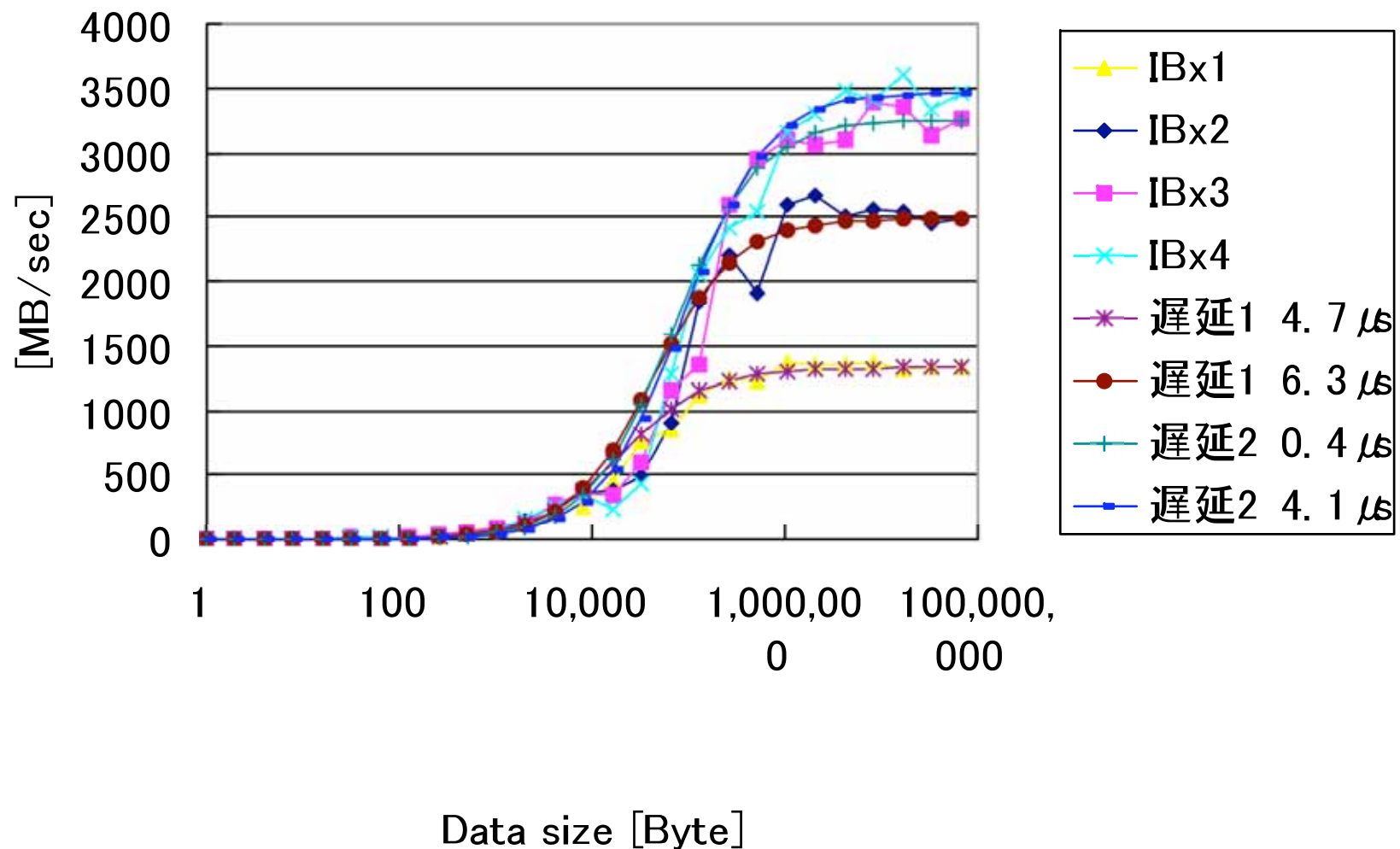
PingPong





[環境2] 遅延、バンド幅での曲線と の比較

PingPong





[環境2] PingPongベンチマークの 考察

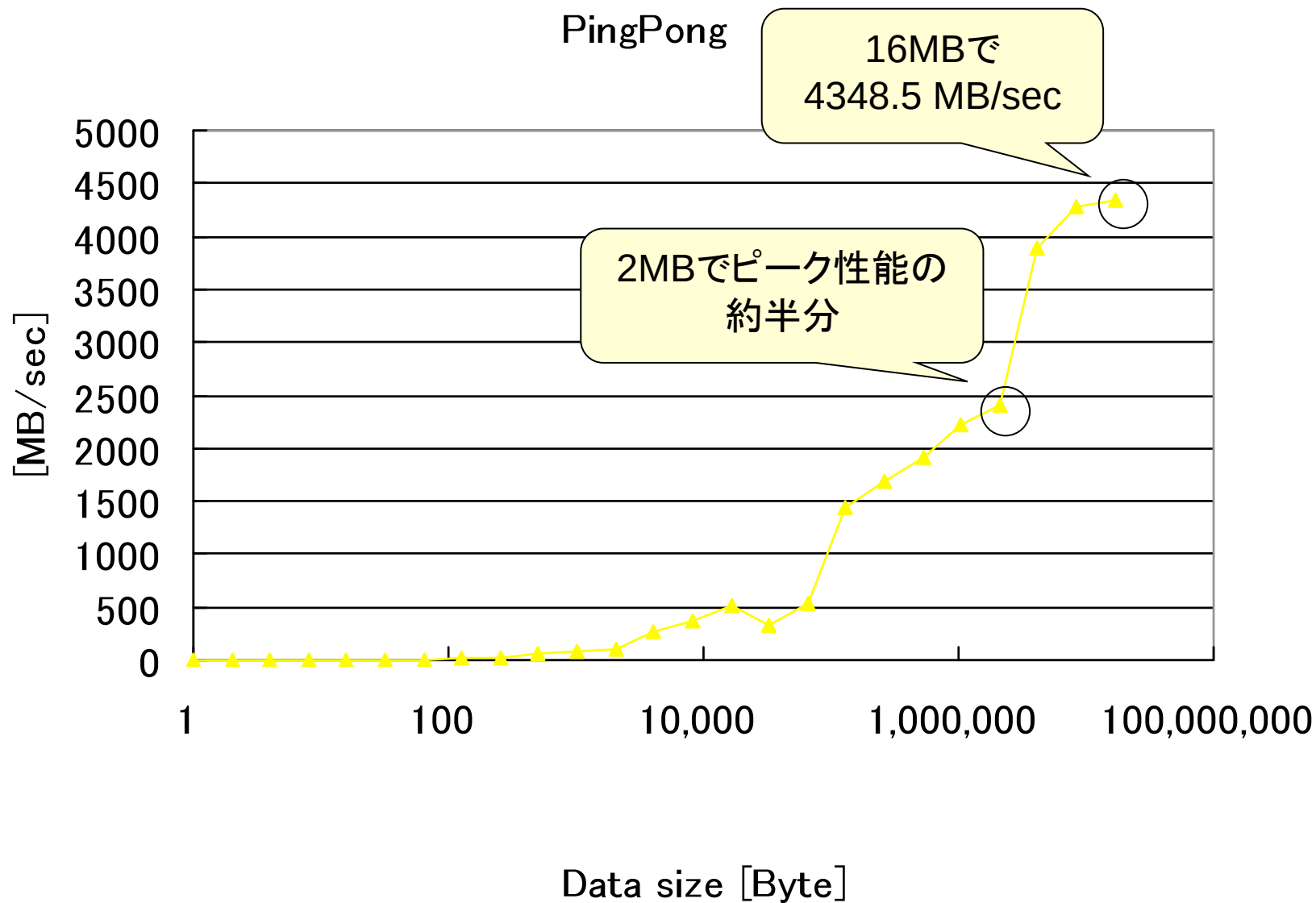
- データサイズは大きい方が高性能

IBの数	1	2	3	4
バンド幅[MB/s]	1366	2674	3256	3468
遅延[μ秒]	14.7	16.3	20.4	24.1
N _{half} [KB]	20	42	68	86

- ショートメッセージとロングメッセージで遅延は変わらない



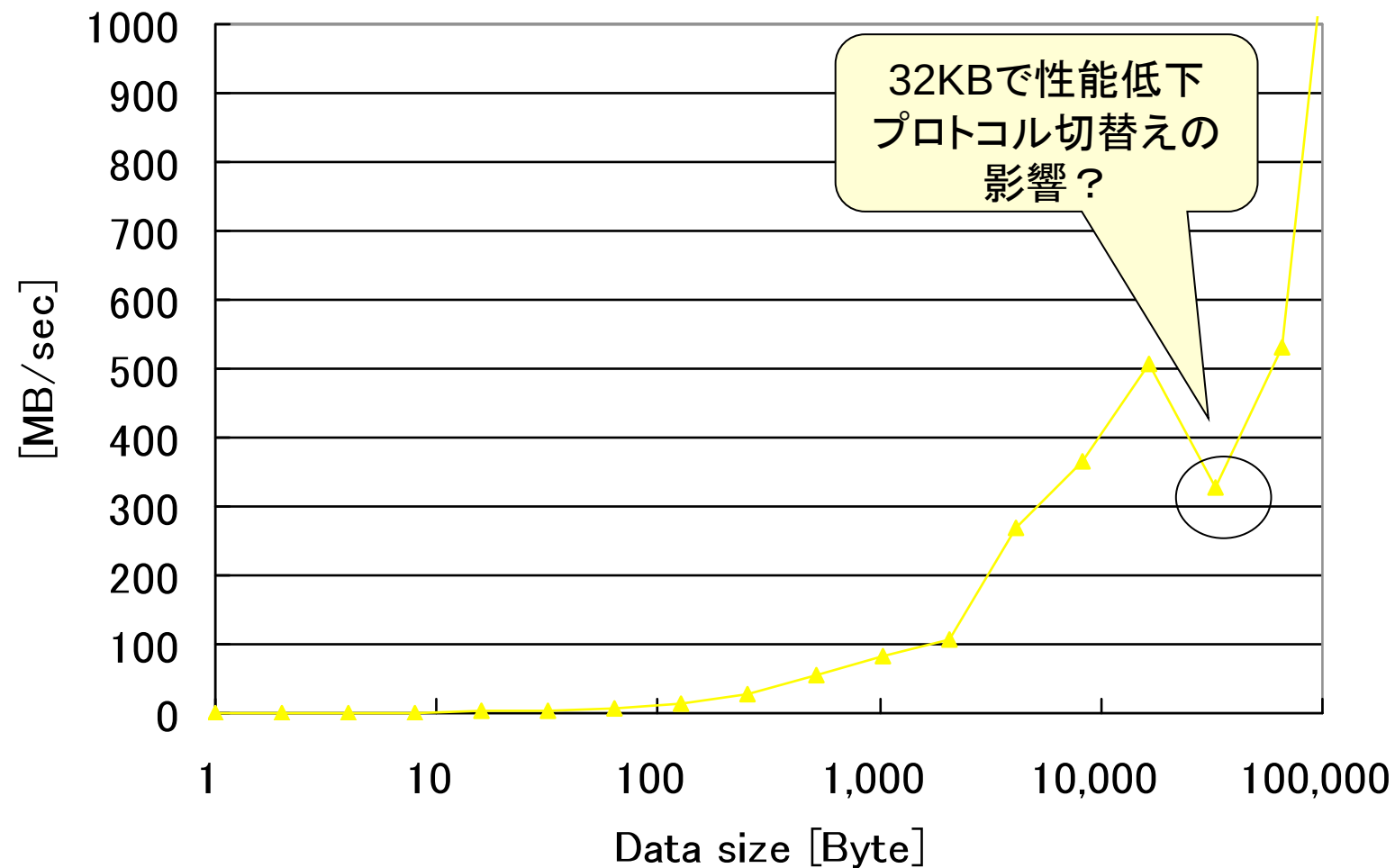
[環境3] PingPongベンチマーク





[環境3] PingPongベンチマーク (拡大)

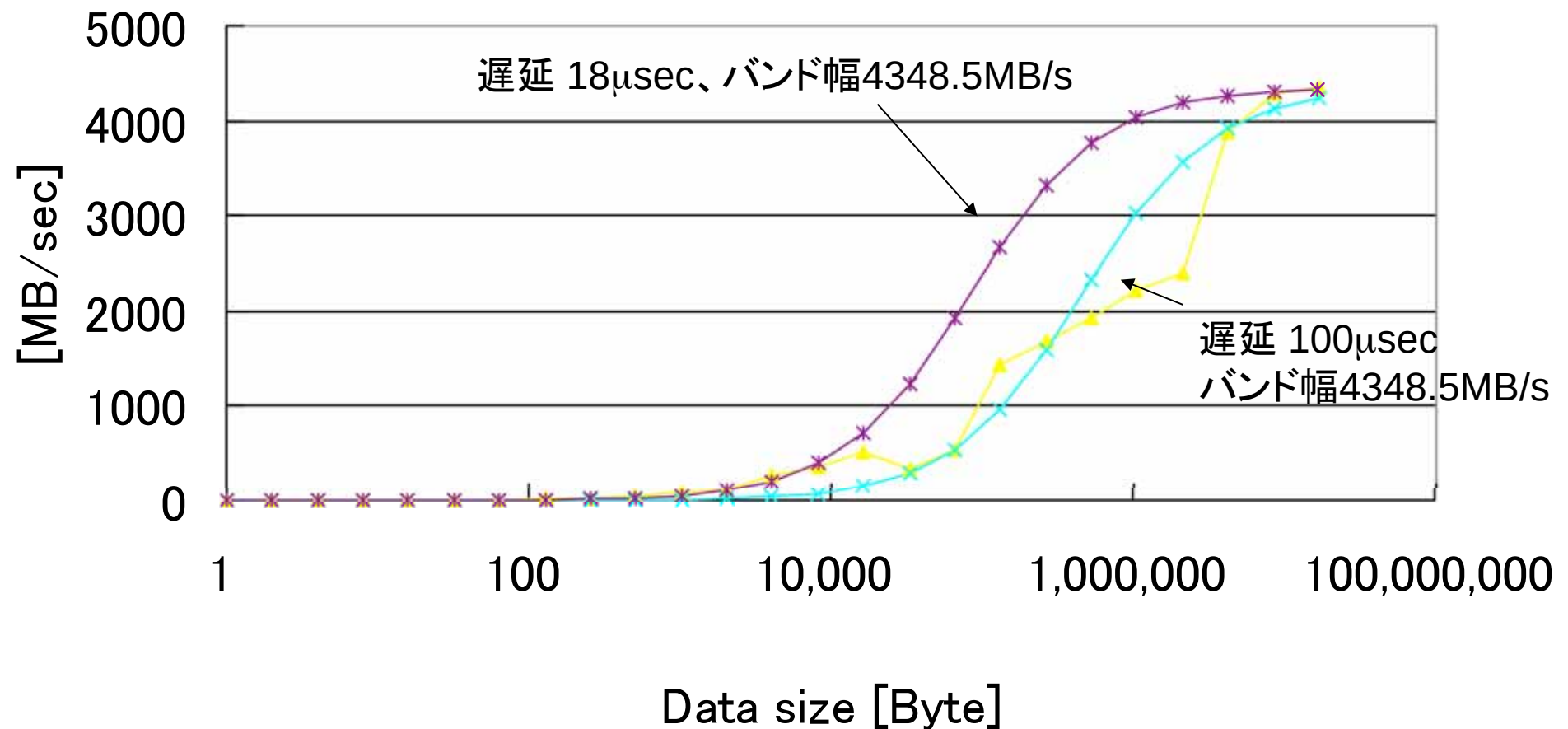
PingPong





[環境3] 遅延、バンド幅での曲線と の比較

PingPong





[環境3] 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
- (マルチレールのためか)特にロングメッセージで遅延は大きめ
- ピークの半分以上→データサイズ2MB以上
- ピークの9割以上→データサイズ4MB以上
- 1バイトのPingPongベンチマークの遅延時間は18msec。ショートメッセージは8~18msec、ロングメッセージは100msec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

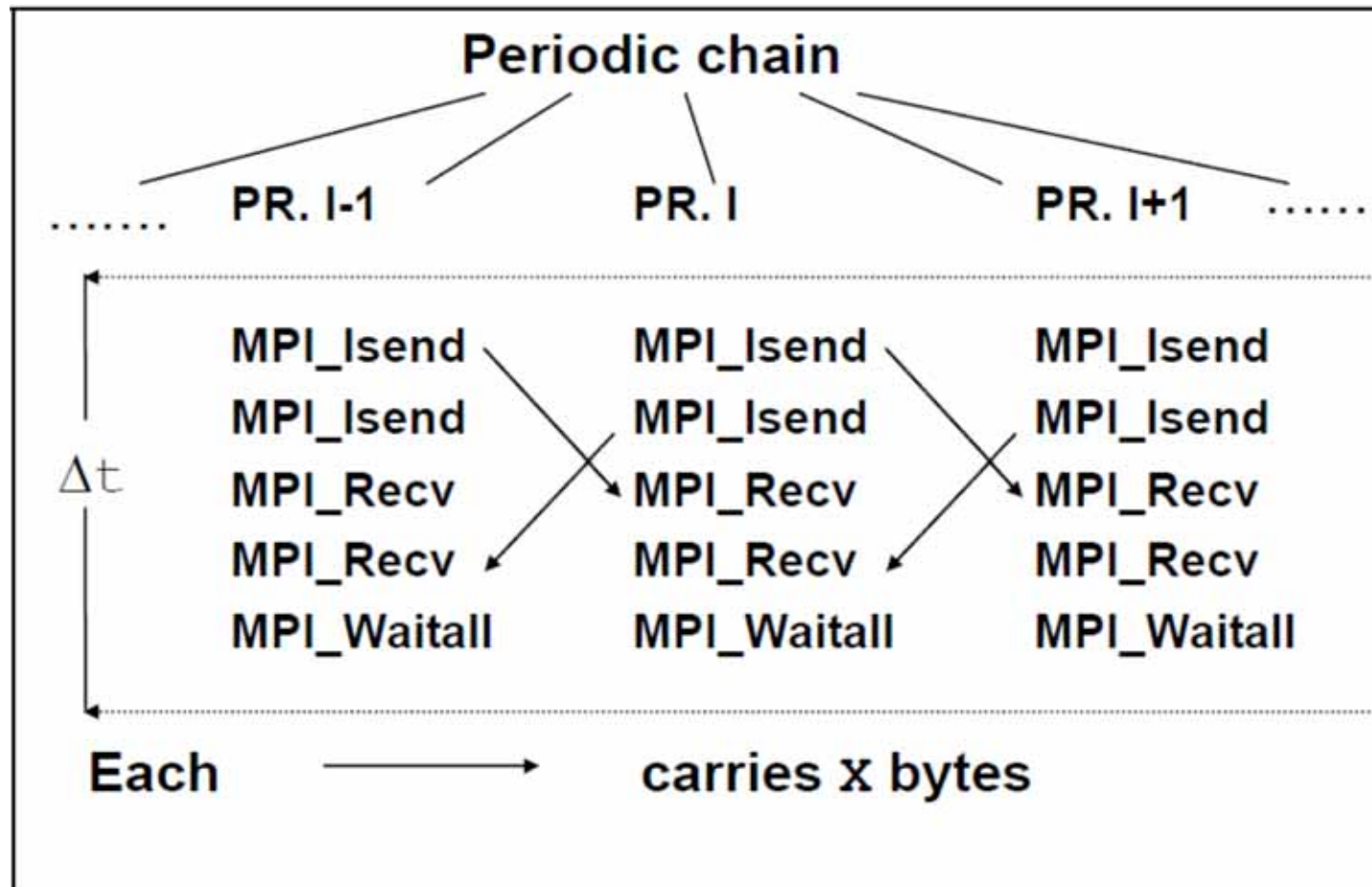
– PingPong	Single Transfer
– PingPing	
– Sendrecv	Parallel Transfer
– Exchange*	
– Bcast	Collective
– Allgather	
– Allgatherv	
– Alltoall*	
– Alltoallv*	
– Reduce	
– Reduce_scatter	
– Allreduce*	
– Barrier	
– 上記を複数一斉に行うMulti版	

- 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}



Exchangeパターン

- 境界の要素を交換する通信パターン

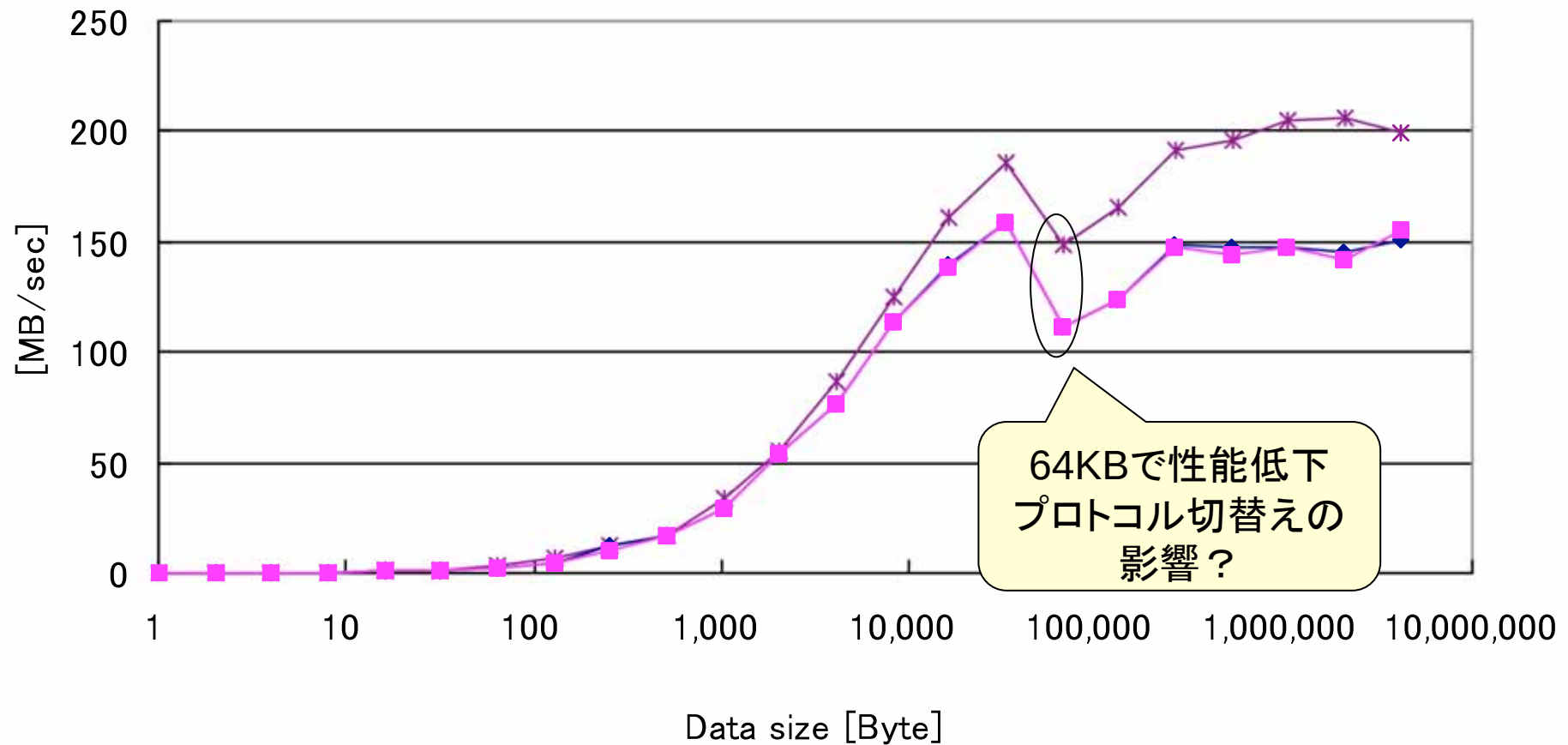


*Intel MPI Benchmarks Users Guide and Methodology Descriptionより



[環境1] Exchange (2ノード) [試行3回]

Exchange (2nodes)



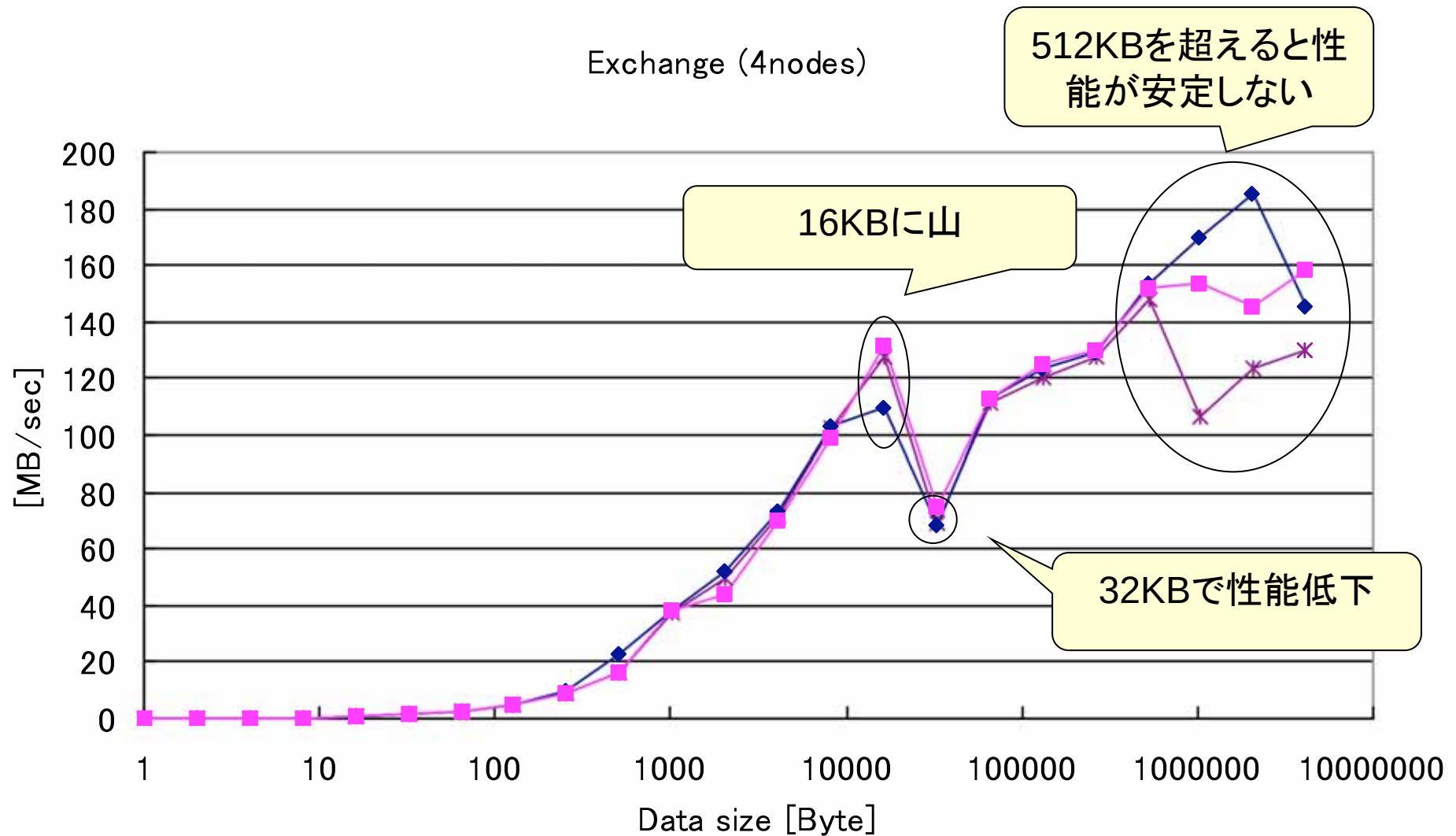


[環境1] Exchange (2ノード) の考察

- データサイズは基本的には大きい方が高性能だが、64KB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8KB以上
- ただし8KBを超えると性能は安定しない
- データサイズ256KB以上ではピークの8割以上でることもある



[環境1] Exchange (4ノード) [試行3回]



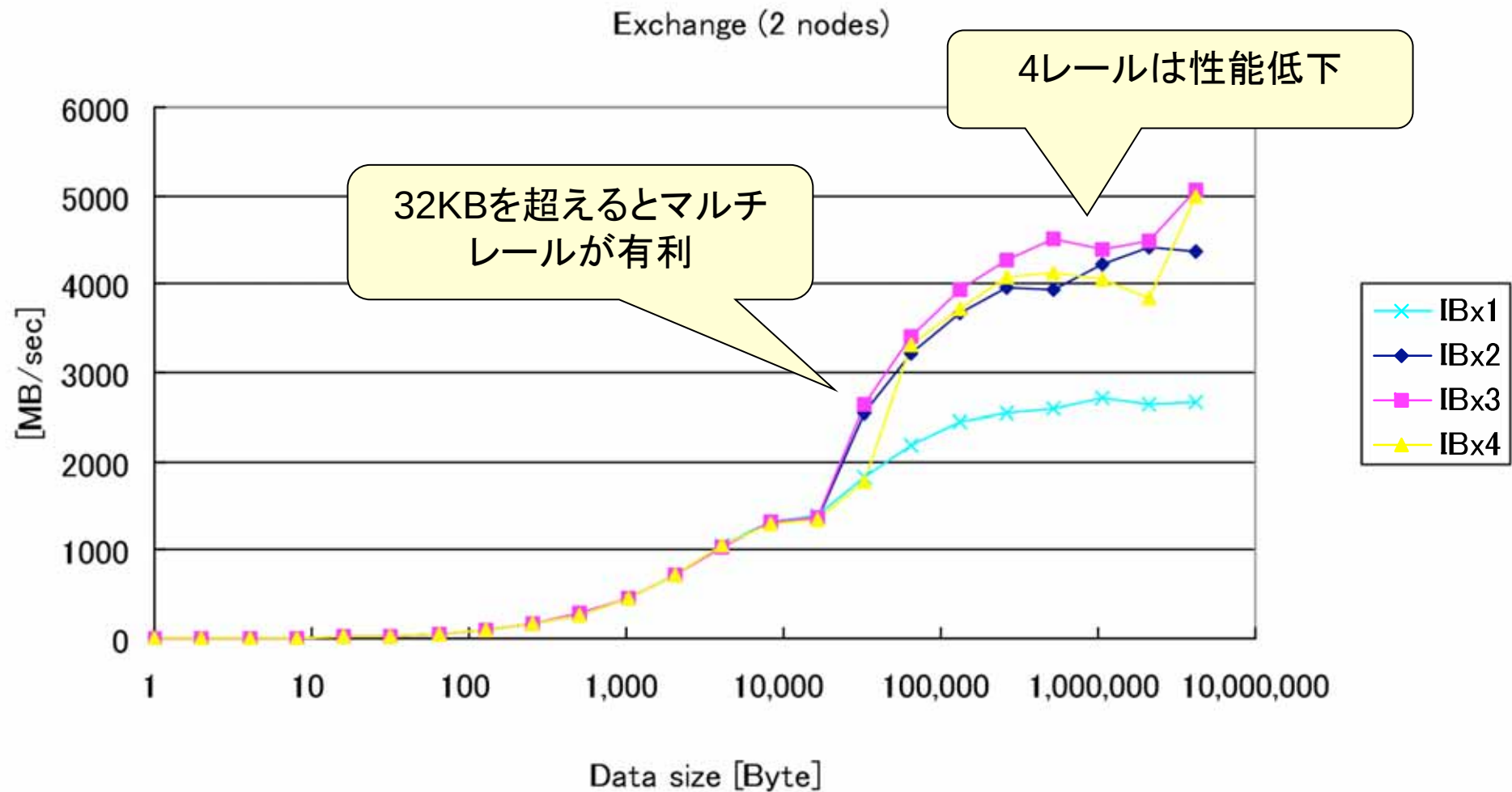


[環境1] Exchange (4ノード) の考察

- 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 $2 \times 113.1 = 226.2 \text{ MB/sec}$
- 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

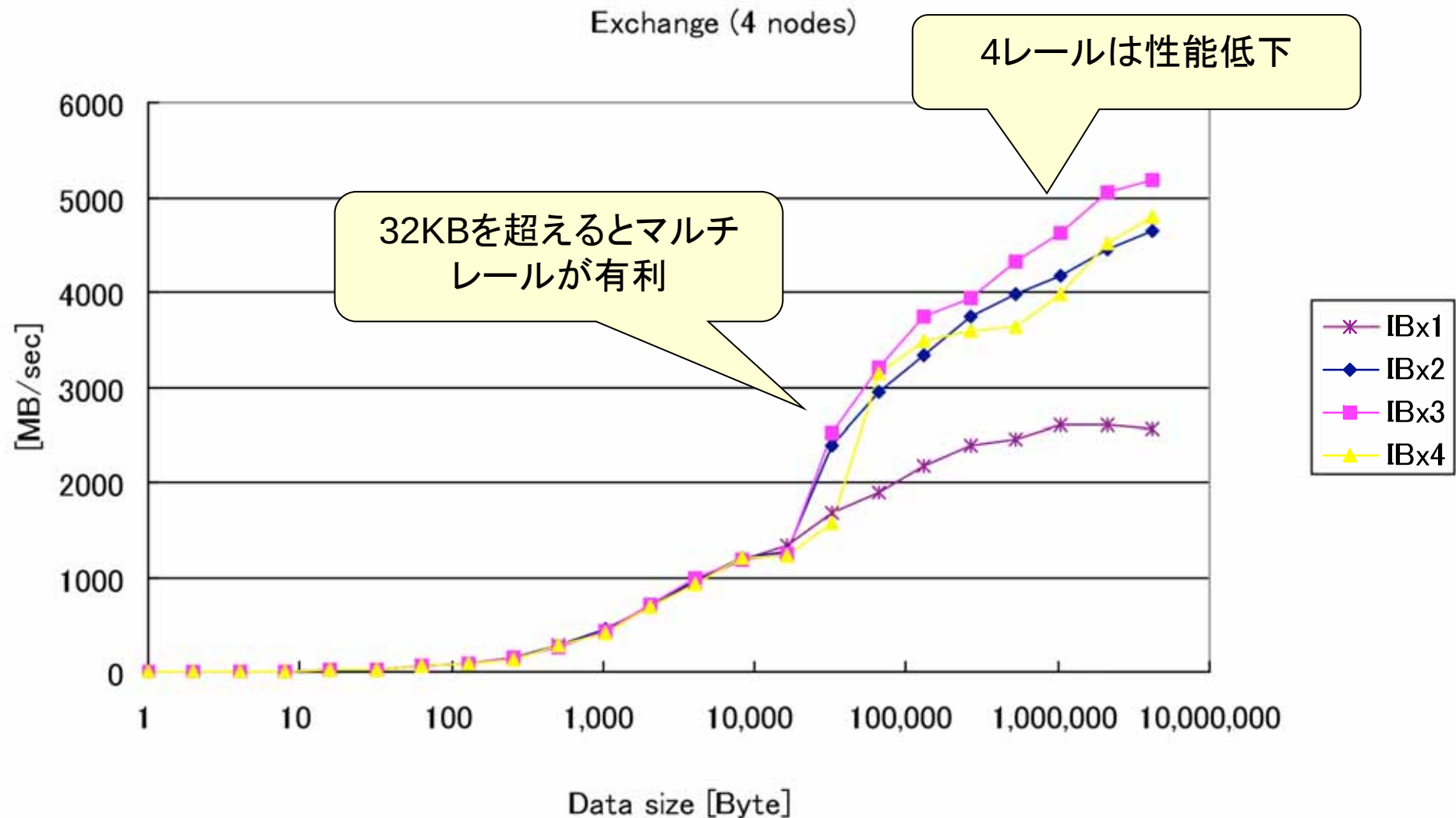


[環境2] Exchange (2ノード)





[環境2] Exchange (4ノード)





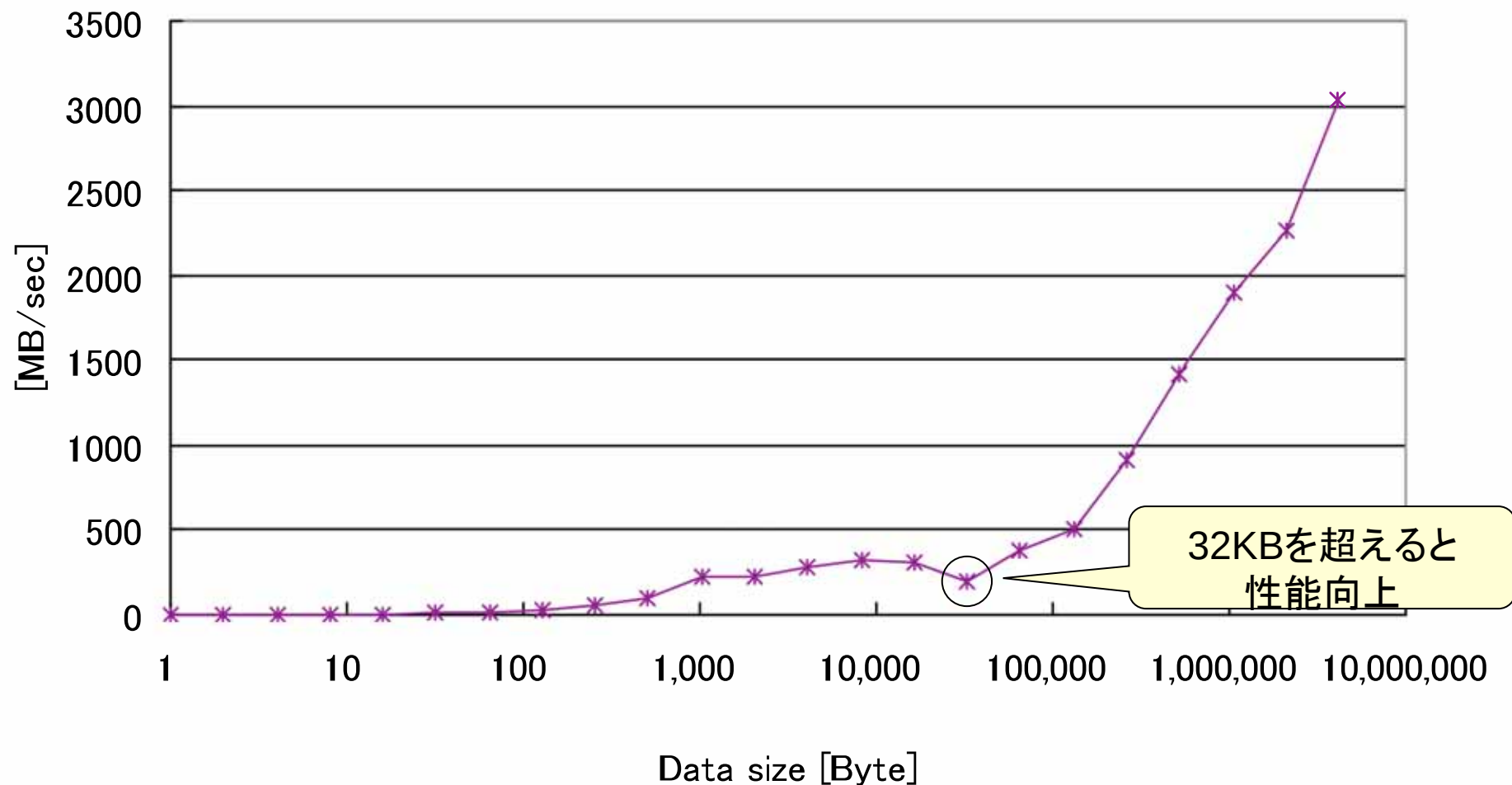
[環境2] Exchangeの考察

- データサイズは基本的には大きい方が高性能
- 32KBを超えるとマルチレールが有利
- 4レールでは性能低下
- 性能は安定している (IBなのでパケットが落ちないことが影響？)



[環境3] Exchange (2ノード)

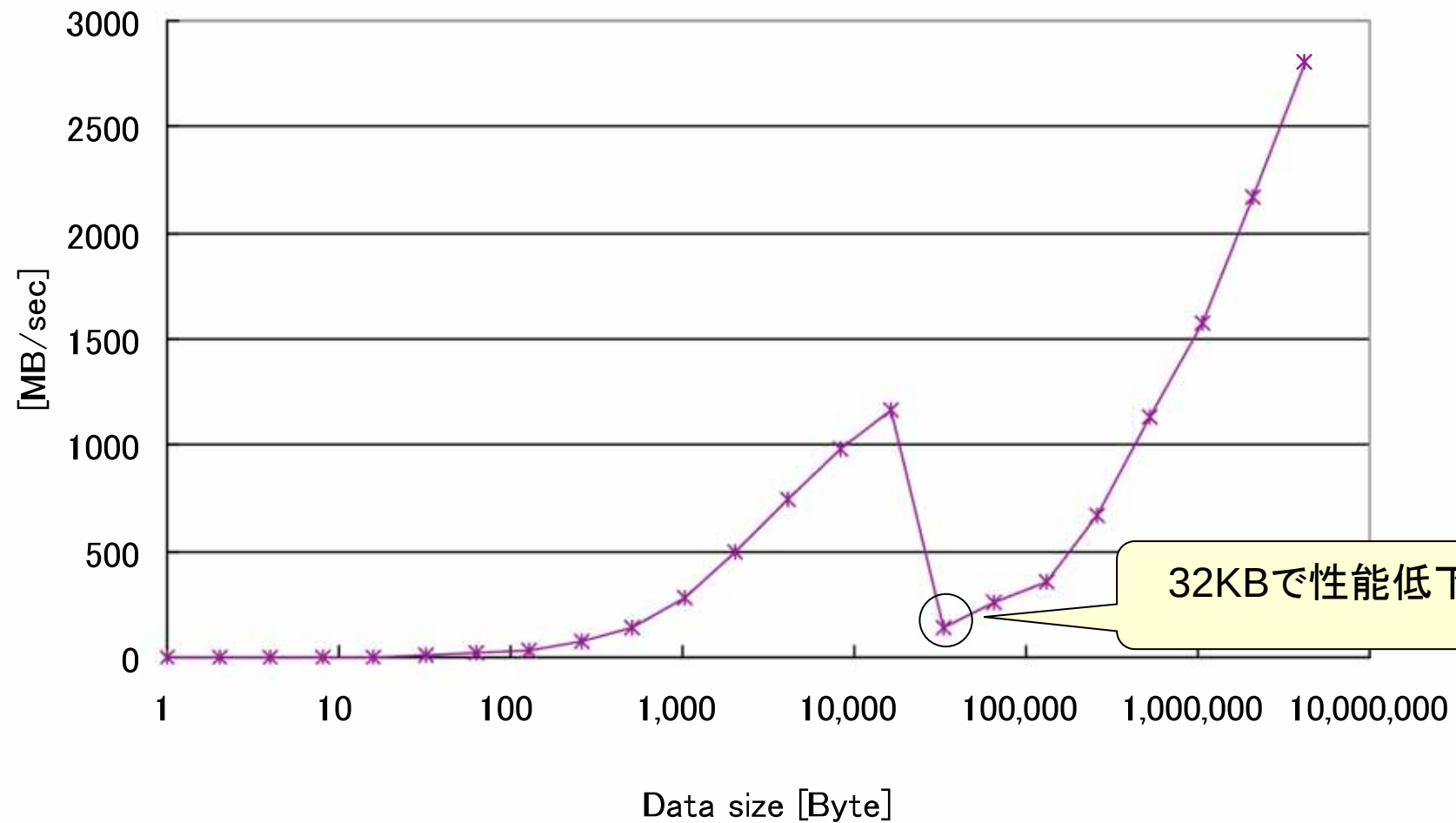
Exchange (2 nodes)





[環境3] Exchange (4ノード)

Exchange (4 nodes)





[環境3] Exchange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
- 32KBを超えると性能が向上
- 性能は安定している（IBなのでパケットが落ちないことが影響？）



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

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

プロセス1
の配列

プロセス2
の配列

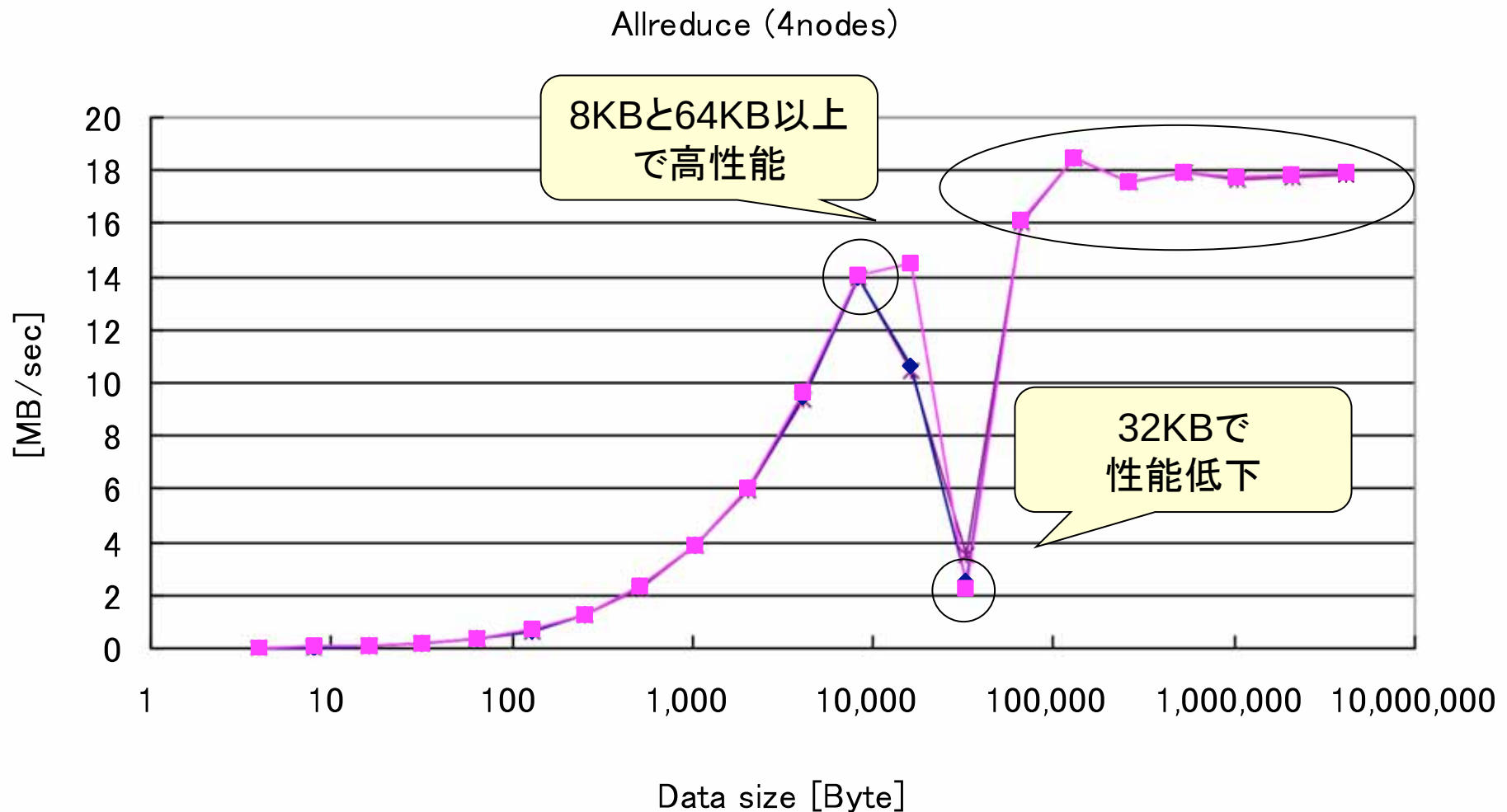
プロセス3
の配列

プロセス4
の配列

計算結果を
全プロセスで保持



[環境1] Allreduce (4ノード) [サイズ/時間]



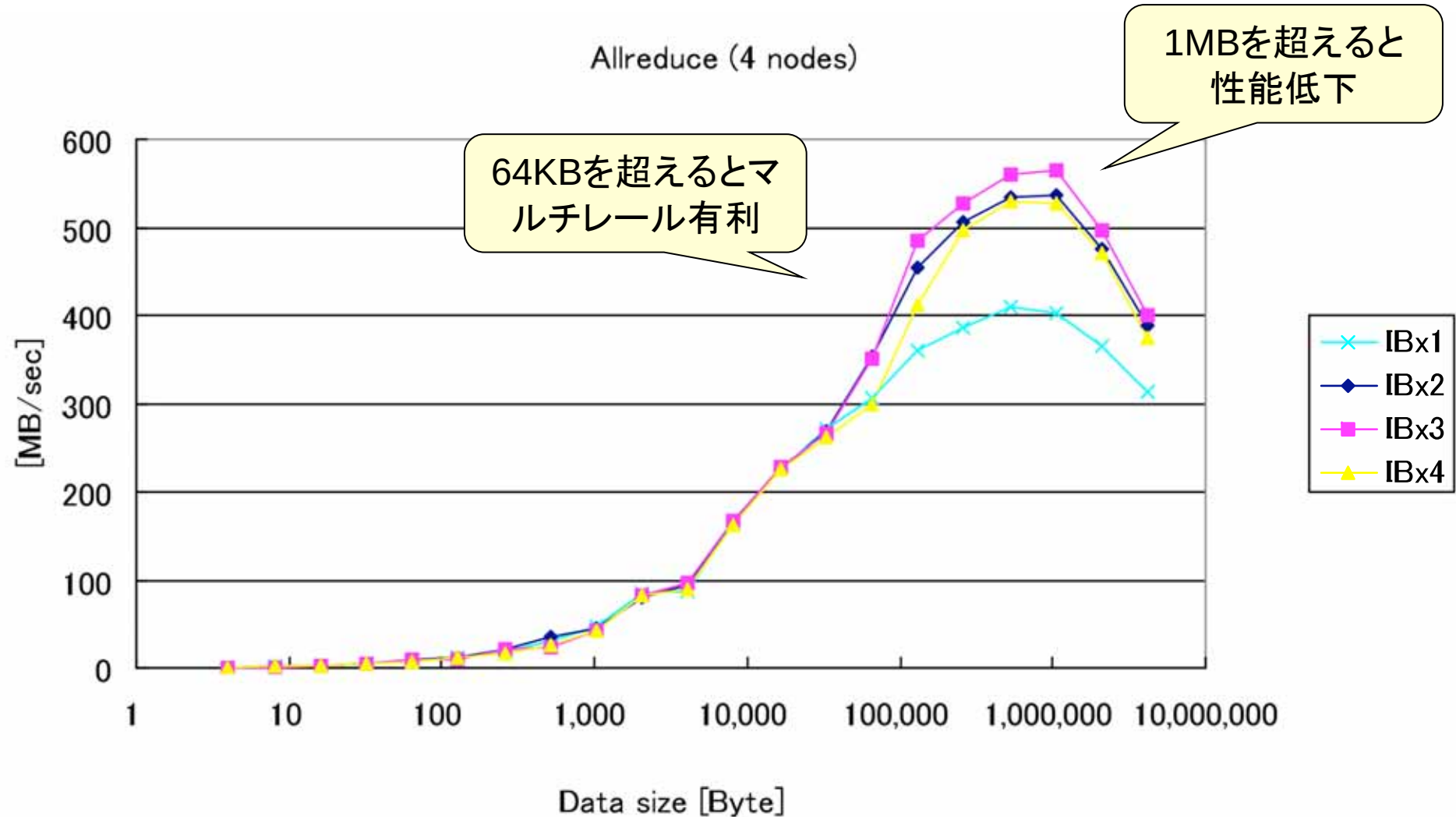


[環境1] Allreduceの考察

- 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]





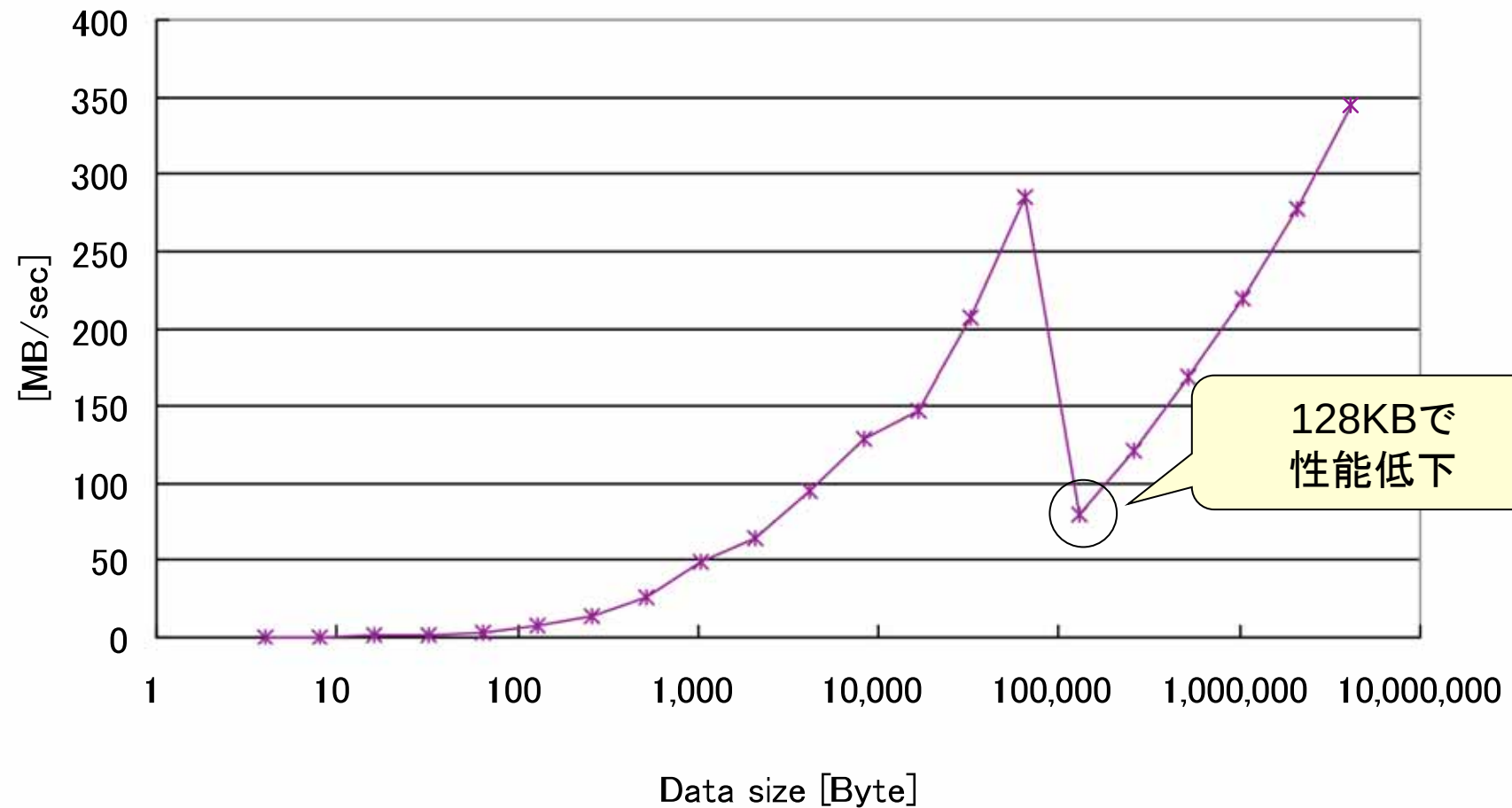
[環境2] Allreduceの考察

- データサイズは基本的には大きい方が高性能だが、1MBを超えると性能低下
- 64KBを超えるとマルチレールが有利
- 4レールは性能低下



[環境3] Allreduce (4ノード) [サイズ/時間]

Allreduce (4 nodes)





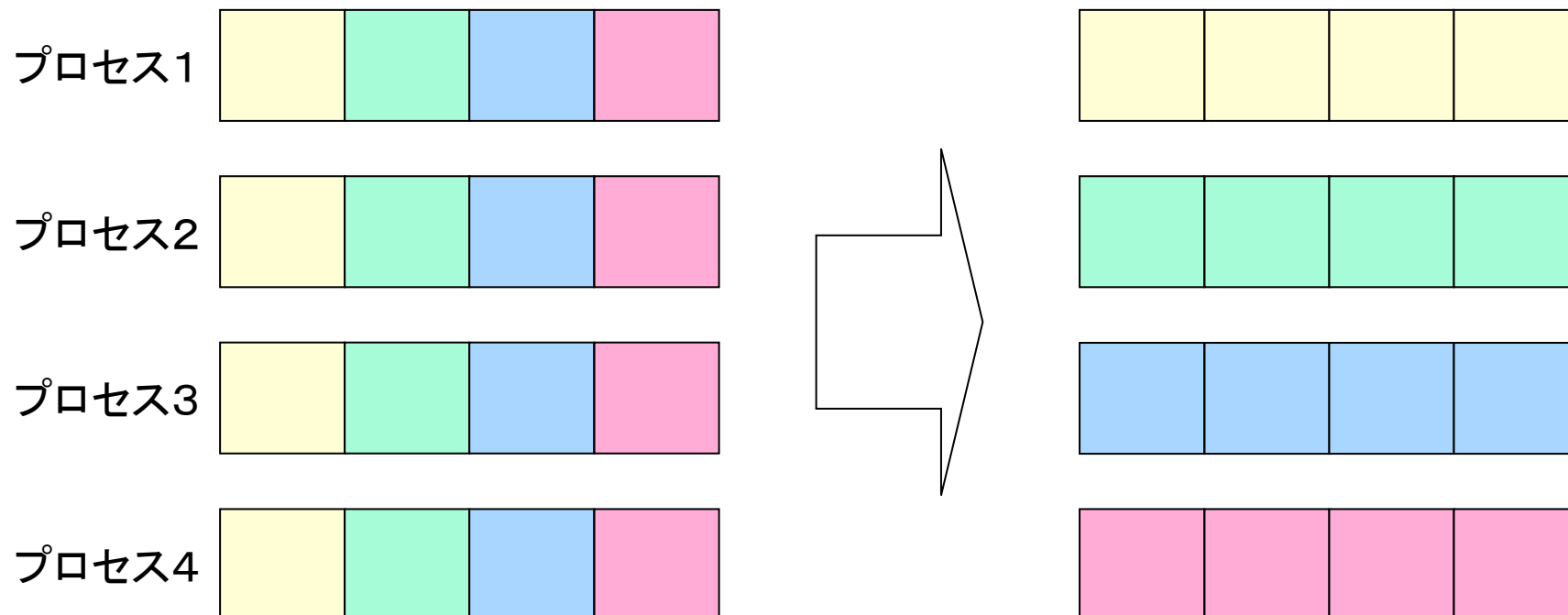
[環境3] Allreduceの考察

- データサイズは基本的には大きい方が高性能だが、128KBで性能低下



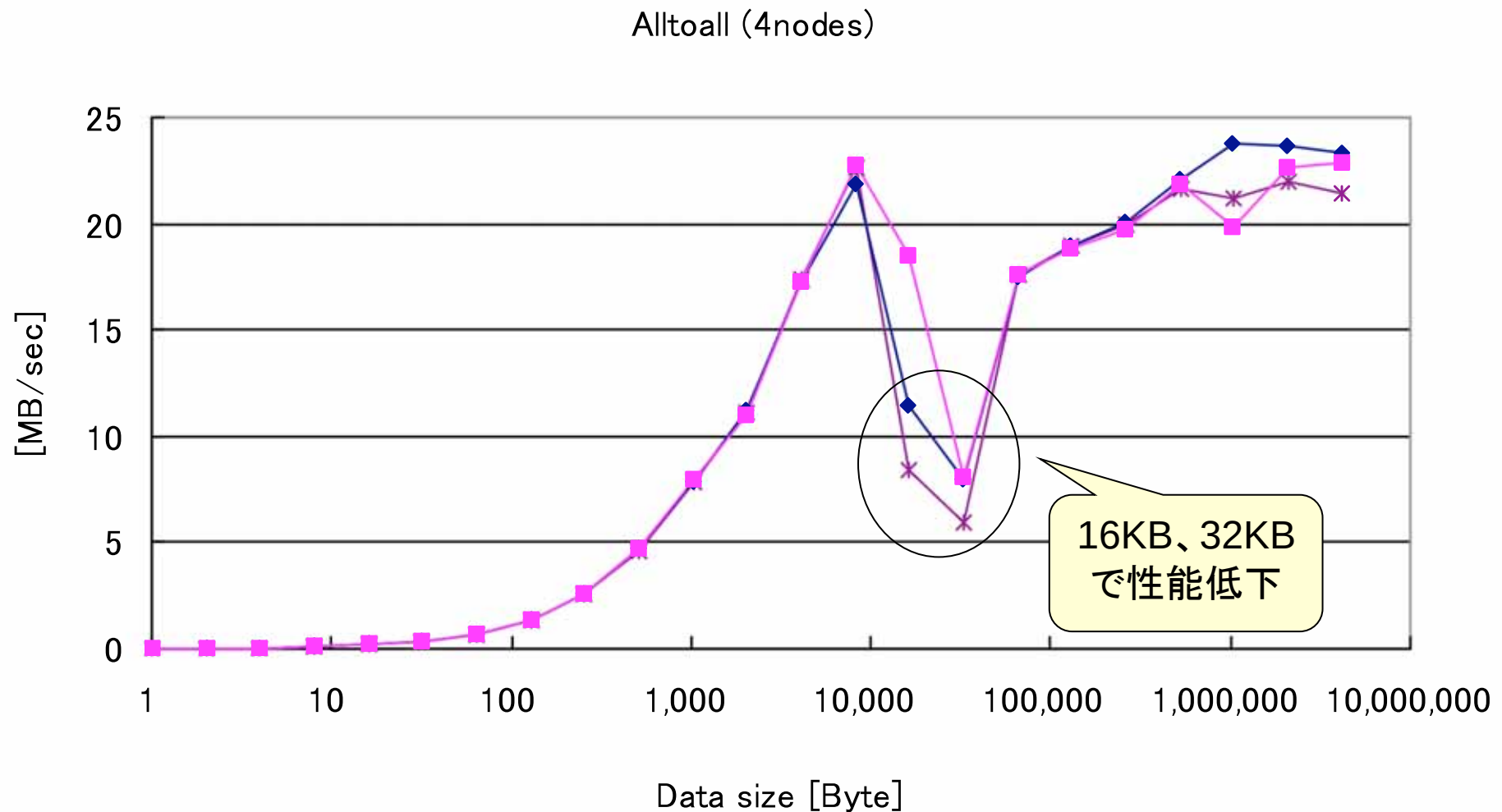
Alltoall

- 行列の転置に相当する集団通信





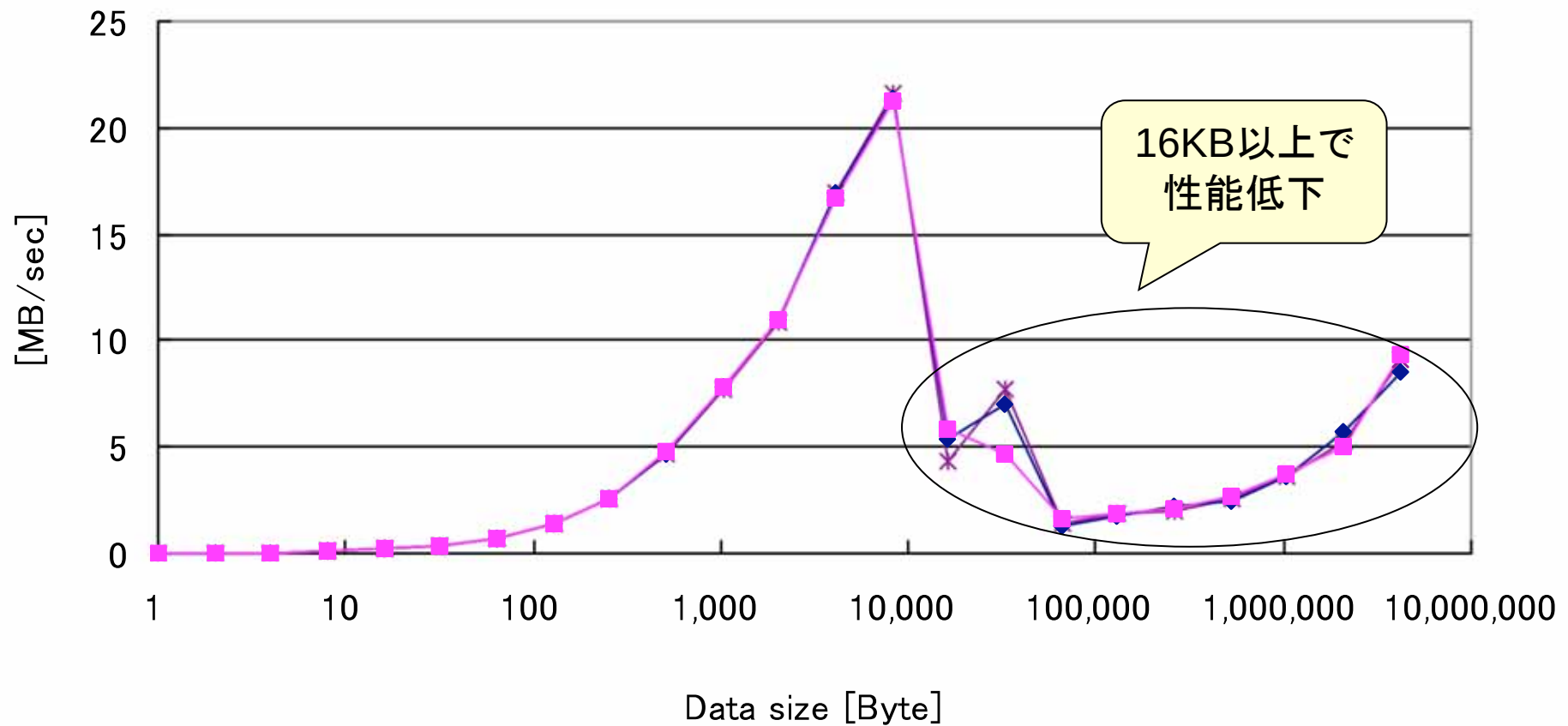
[環境1] Alltoall [サイズ/時間]





[環境1] Alltoallv [サイズ/時間]

Alltoallv (4nodes)



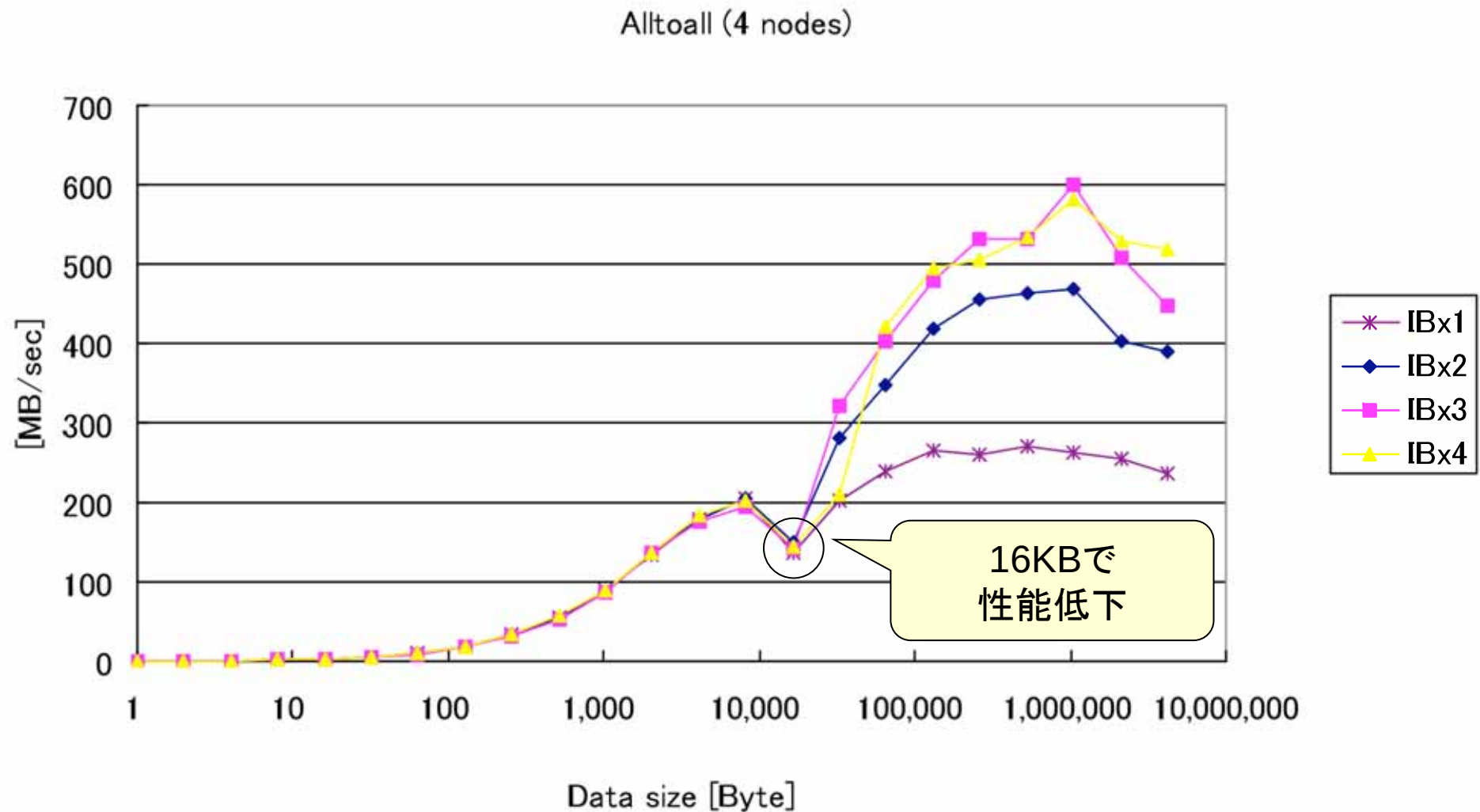


[環境1] Alltoall(v)の考察

- Alltoallは、データサイズは基本的には大きい方が高性能だが、16KB～32KBで性能低下
 - 8KB、64KB以上では高性能
 - Allreduceと同様
- Alltoallvは、16KBを超えると極端に性能低下
 - 不必要なメモリコピーのためと思われる
 - 性能最適化がなされていない

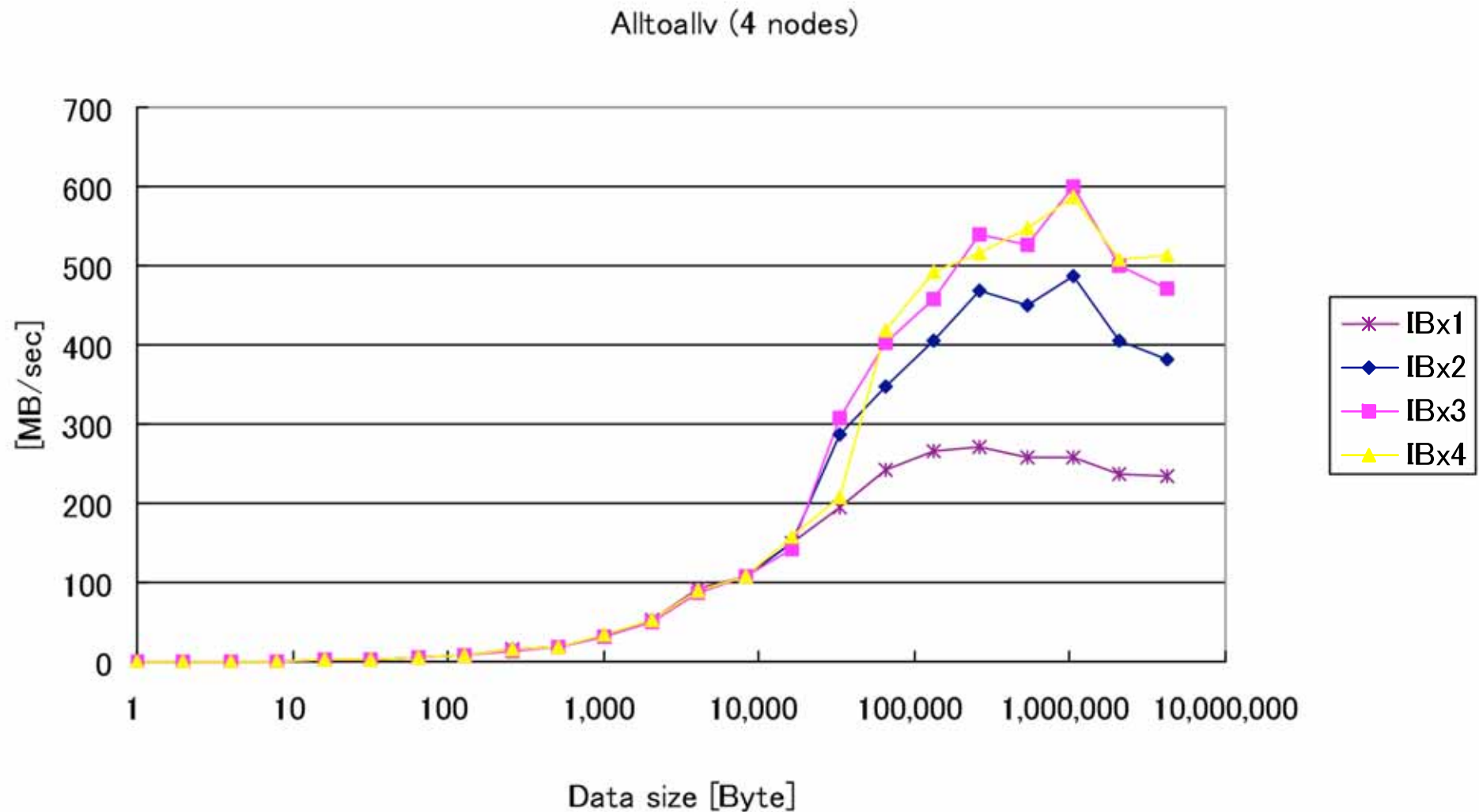


[環境2] Alltoall [サイズ/時間]





[環境2] Alltoallv [サイズ/時間]



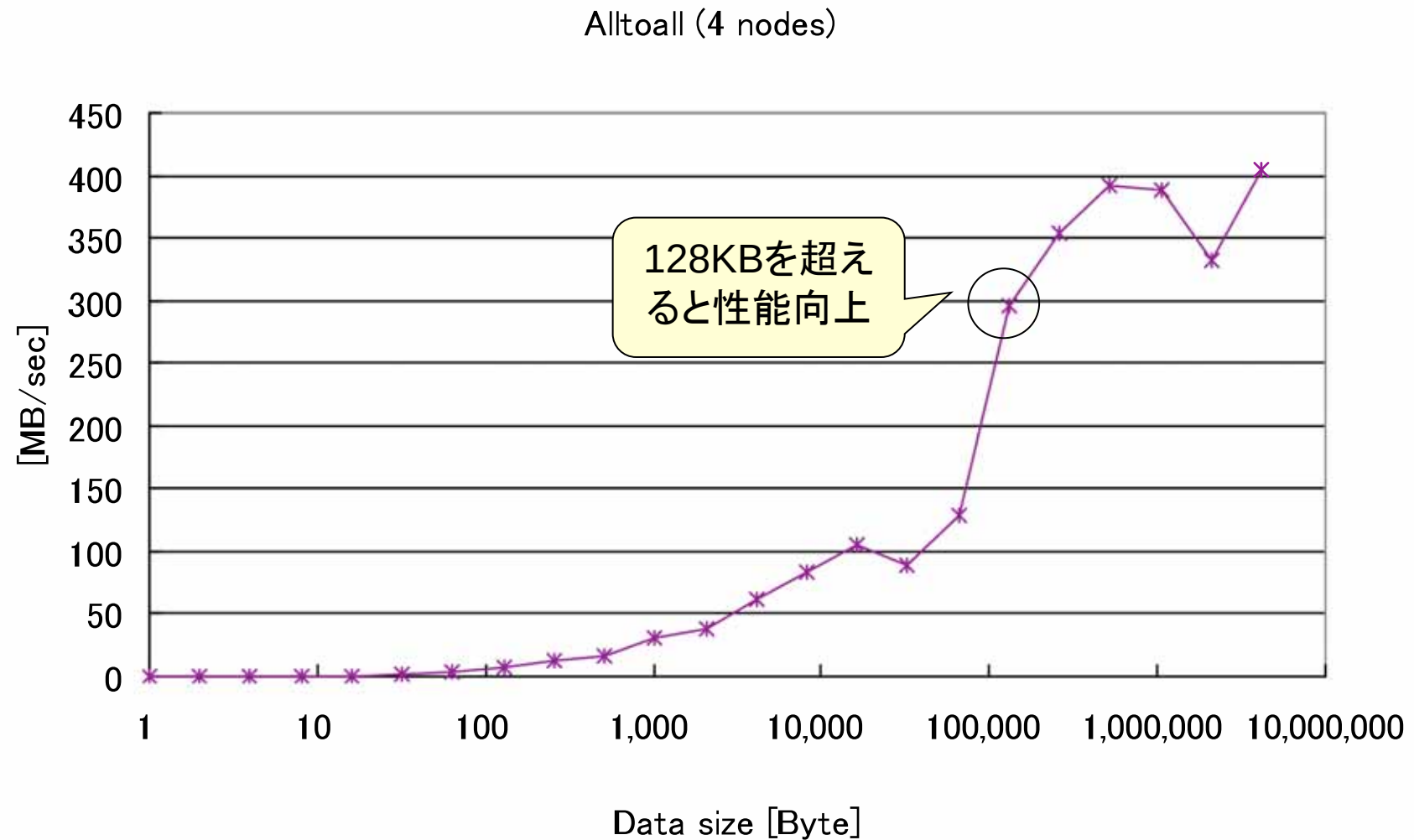


[環境2] Alltoall(v)の考察

- Alltoall, Alltoallvは、データサイズは基本的には大きい方が高性能
- Alltoallは16KBで性能低下
- 32KBを超えるとマルチレールが有利



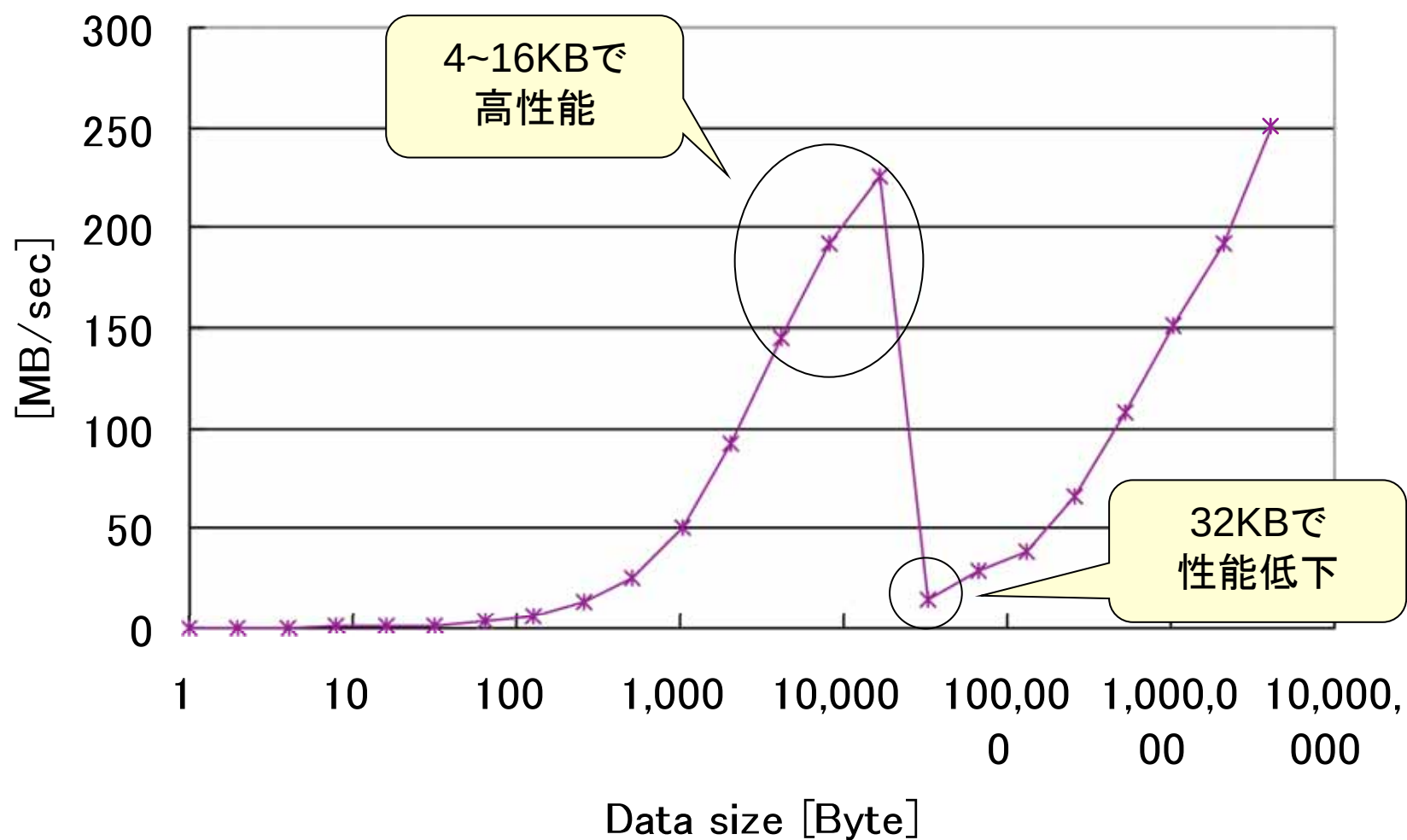
[環境3] Alltoall [サイズ/時間]





[環境3] Alltoallv [サイズ/時間]

Alltoallv (4 nodes)





[環境3] Alltoall(v)の考察

- Alltoallは、データサイズは基本的には大きい方が高性能。特に128KBを超えると性能向上
- Alltoallvは、4~16KBで高性能であるが、32~256KBで性能低下。その後性能は向上する



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



tlog – time log

- 実行プロファイルをとるための軽量ライブラリ(佐藤センター長 @筑波大)
 - 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://www.ccs.tsukuba.ac.jp/workshop/HPCseminar/2008/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

- π 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

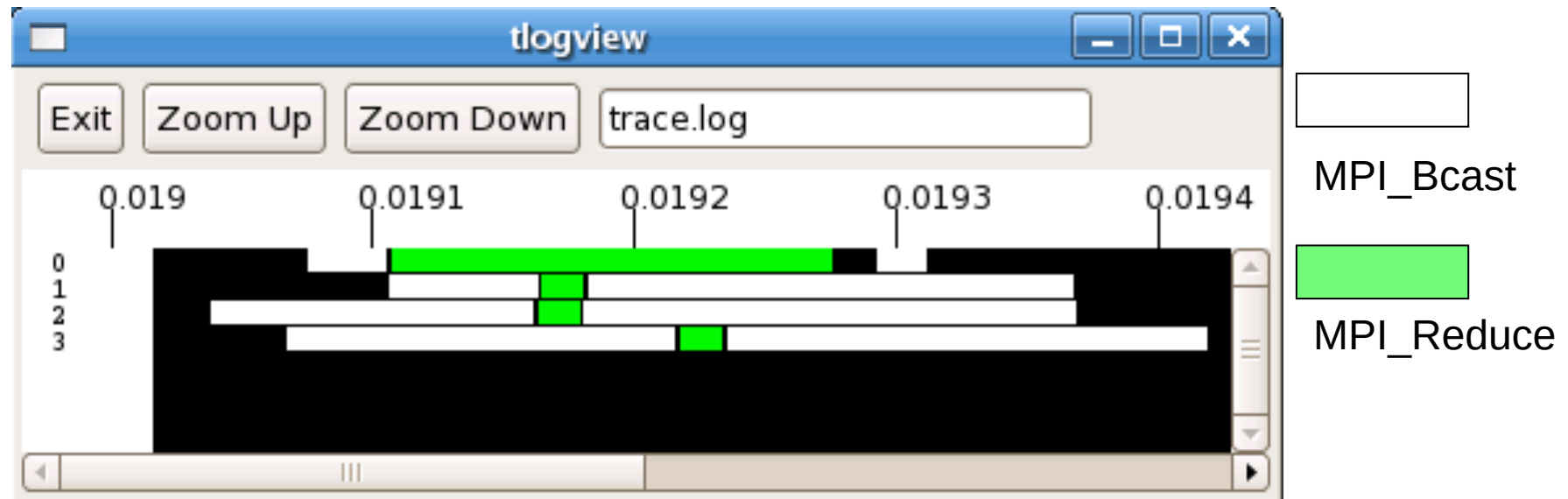


cpiのプロファイル結果(1)

- 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

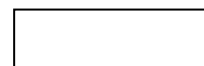
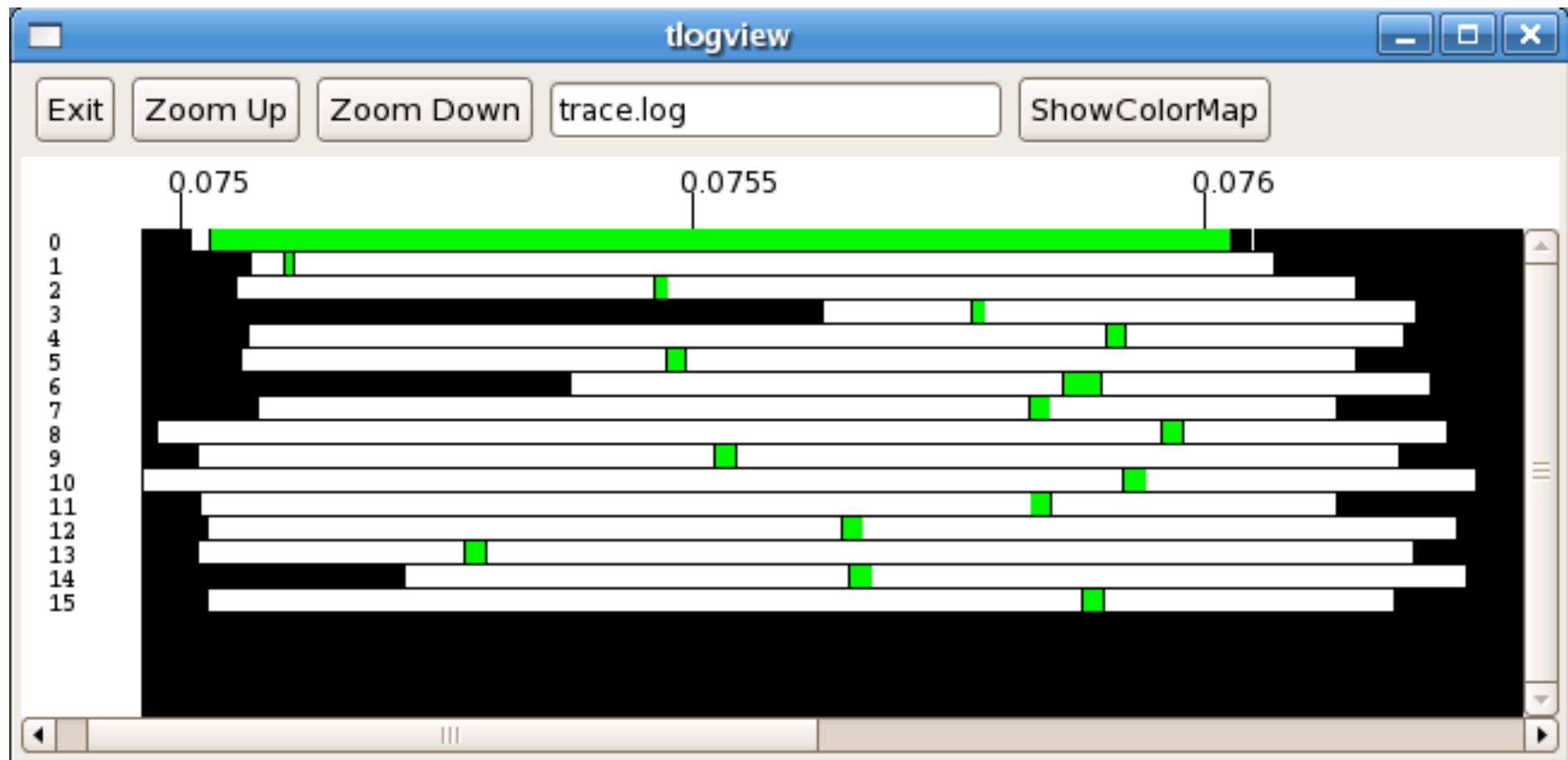


tlog_initializeからの経過時間(秒)。ノード間の時刻差修正済。



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*
- 通信遅延隠蔽
- 通信ブロック
- 負荷分散*



通信ブロッキング

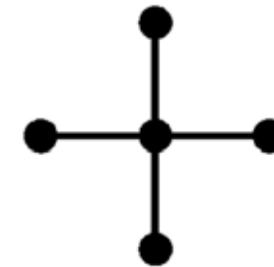
- 1対1通信、集団通信はデータサイズによって通信性能が大きく変化する
- 通信ブロッキングは、通信データをまとめてデータサイズを変更する(大きくする)手法
 - データのブロック分散
 - 複数反復をまとめる



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 */  
  }  
}
```

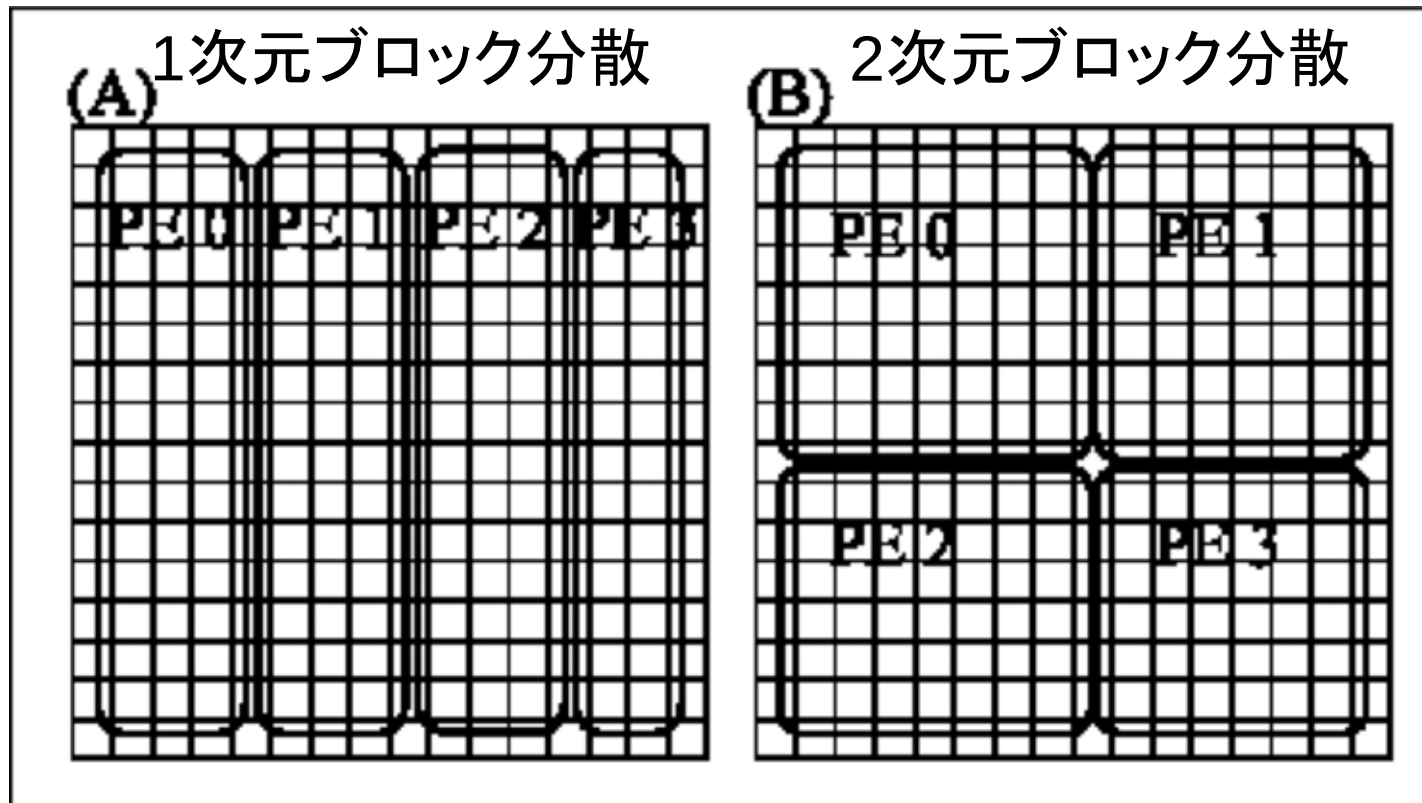


データ依存関係

*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 $n / \sqrt{p}$



複数反復をまとめる

- ヤコビ法二反復をまとめる

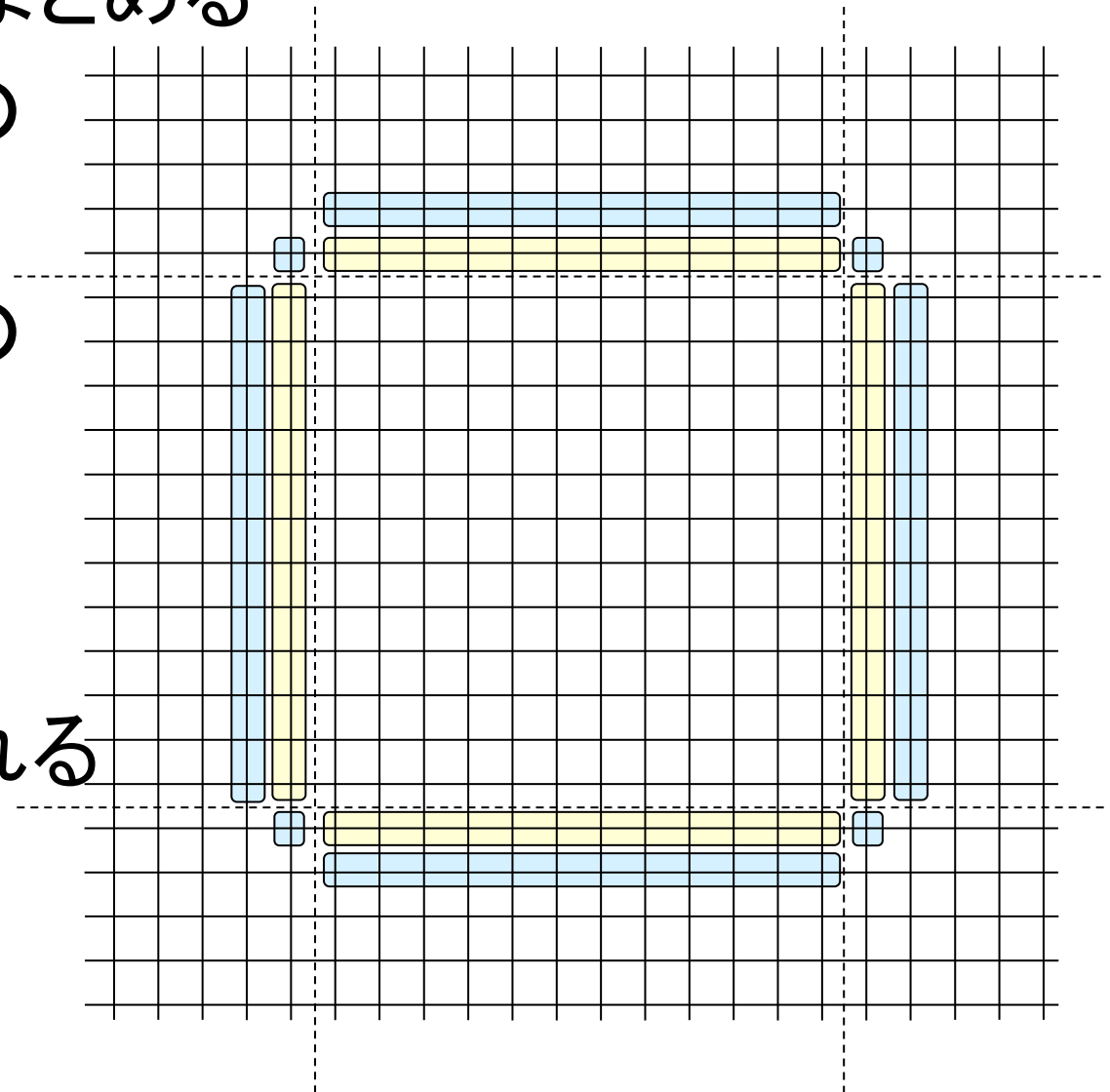
- 一反復目では  のデータが必要

- 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 $2n / \sqrt{p}$





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。