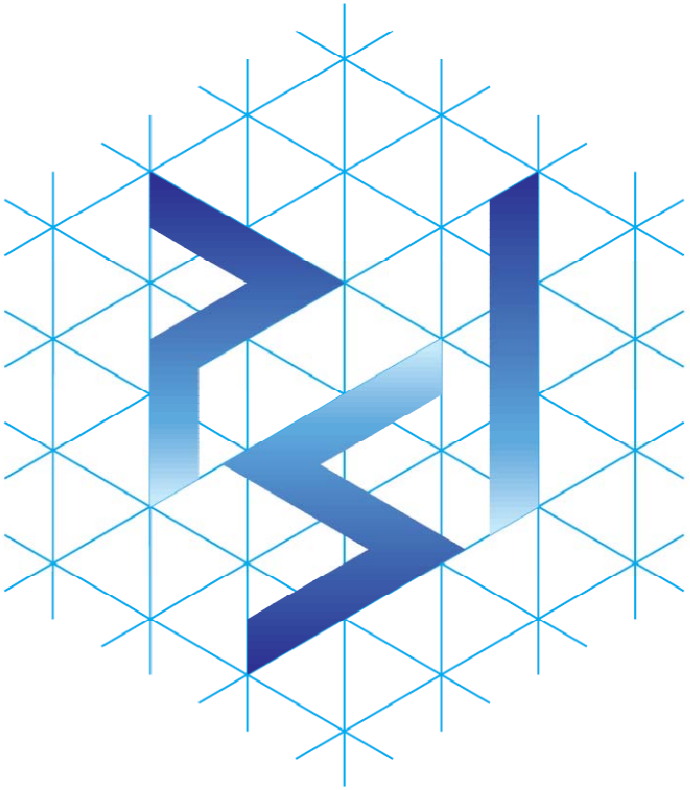
ペタスケールインターコネクトに向けたMPI高速化技術の研究

南里, 豪志
九州大学情報基盤研究開発センター

出版情報：SLRC プレゼンテーション, 2007-07-25. 九州大学システムLSI研究センター
バージョン：
権利関係：

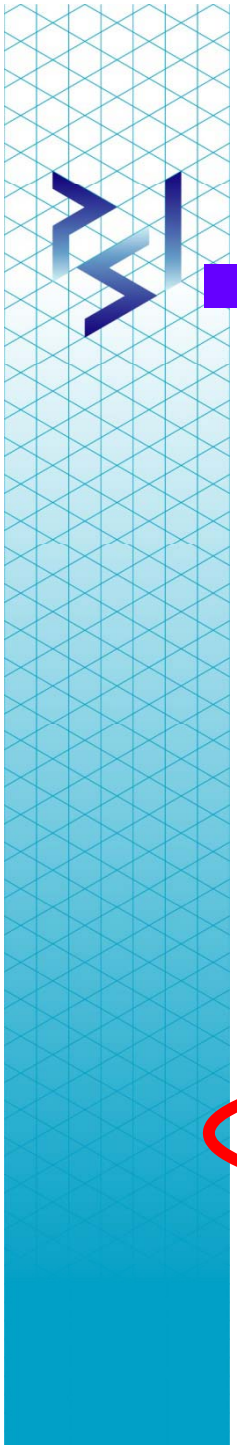
ペタスケールインターコネクトに向けた MPI高速化技術の研究



2007年 7月25日

九州大学　南里 豪志

PETASCALE SYSTEM INTERCONNECT PROJECT



PSIプロジェクト

- 文部科学省「次世代IT基盤構築のための研究開発」（研究開発領域「将来のスーパーコンピューティングのための要素技術の研究開発」）
  ペタスケール・システムインターコネクト技術の開発
  - 数1000～数10000台規模の計算ノードを相互接続するシステムインターコネクトの高機能化、高性能化
  - サブテーマ1：光パケットスイッチの実現を目指した物理層技術
  - サブテーマ2：MPIから物理層までを通したインターコネクト全体の高機能化、高性能化技術
  - サブテーマ3：ペタフロップス級マシンの振る舞いをシミュレーション可能とした統合型システム性能評価技術

1分でわかるMPI

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int n = 1000;
  int myid, numprocs, i;
  double mypi, pi, h, sum, x;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  h = 1.0 / (double) n;  sum = 0.0;
  for (i = myid + 1; i <= n; i += numprocs){
    x = h * ((double)i - 0.5);
    sum += (4.0 / (1.0 + x*x));
  }
  mypi = h * sum;
  MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (myid == 0)
    printf("pi is approximately %.16f¥n", pi);
  MPI_Finalize();
  return 0;
}
```

- MPI (Message Passing Interface) プログラム
  - 文法は、普通の C言語 / Fortran
  - 適宜MPI関数を利用して並列処理
  - SPMD (Single Program Multiple Data)

MPI環境初期化

プロセス数の取得

プロセスの番号（＝ランク）の取得

並列計算（ランクに応じて計算）
- ランク 0： i = 1, 5, 9, ...
- ランク 1： i = 2, 6, 10, ...
- ランク 2： i = 3, 7, 11, ...
- ランク 3： i = 4, 8, 12, ...

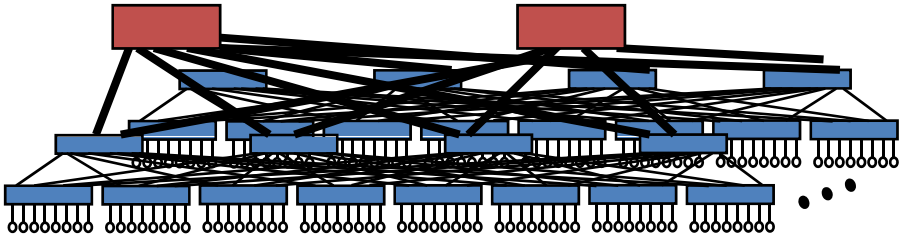
全プロセスのmypiの総和を計算して pi に格納

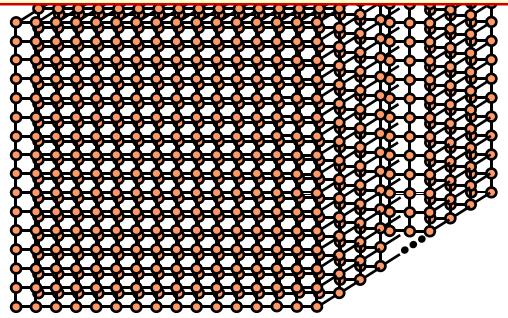
MPI環境終了

ペタスケールシステムインターコネクトに向けたMPIプログラムの高速化技術

- 数千～数万ノードによる大規模並列計算
  - 不均等な負荷分散, 不均質なネットワーク, 他のジョブの影響
  - 最適化（＝性能チューニング）に必要な情報が, 実行してみるまで分からない
    - 負荷の不均衡による同期ポイントへの到着遅れ
    - ランク配置や通信タイミングによる通信衝突
    - 他のジョブの存在による基本通信性能の低下

本研究： 実行時の性能情報に基づいた通信ライブラリ（MPI）実装の自動最適化





本プロジェクトで開発中の最適化技術

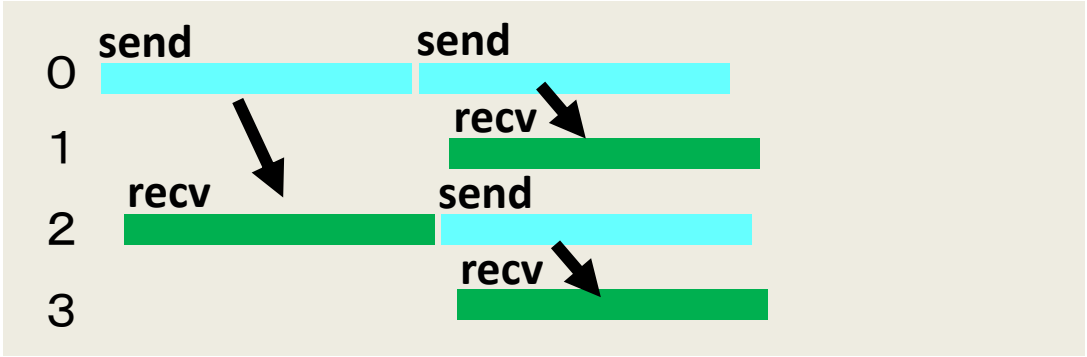
- 各MPI関数の実行タイミングを考慮した最適化
  - 全体通信内部の通信順序調整
    - 曽我 武史（福岡IST）、栗原 康志（九大）
  - ランク配置最適化
    - 森江 善之（九大）、
      Guilherme Domingues（九州ISIT）
- 実行時の性能に応じた全体通信アルゴリズムの選択
  - 性能モデルによる性能予測を利用した最適アルゴリズムの導出
    - Hyacinthe Nzigou Mamadou（九大）、
      Feng Long Gu (九大）、Vivien Odou（九州ISIT)

⇒ Hyacintheさんの発表（本日午後）

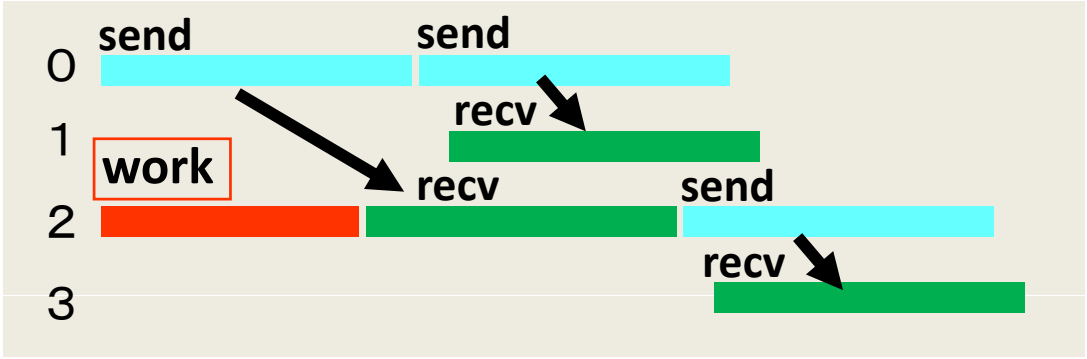
全体通信内部の通信順序調整

MPI_Bcast関数における通信順序の性能への影響

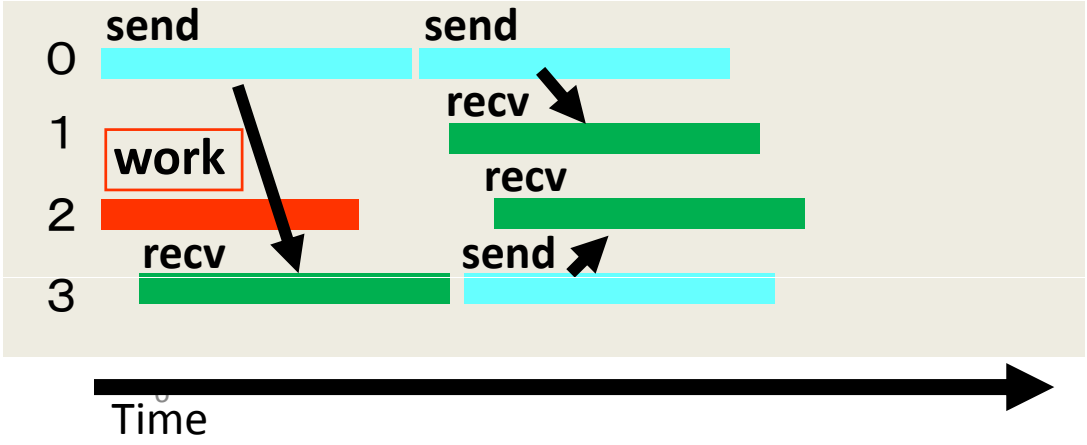
Case 1:
負荷バランス
均等



Case 2:
ランク２が到着遅れ（負荷不均等）

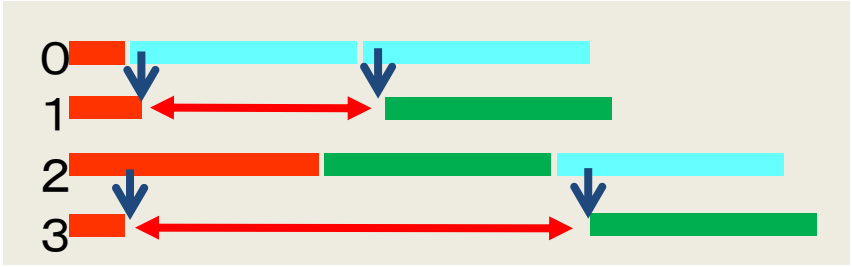


Case 3:
ランク２の遅れを隠ぺいするよう順序を変更



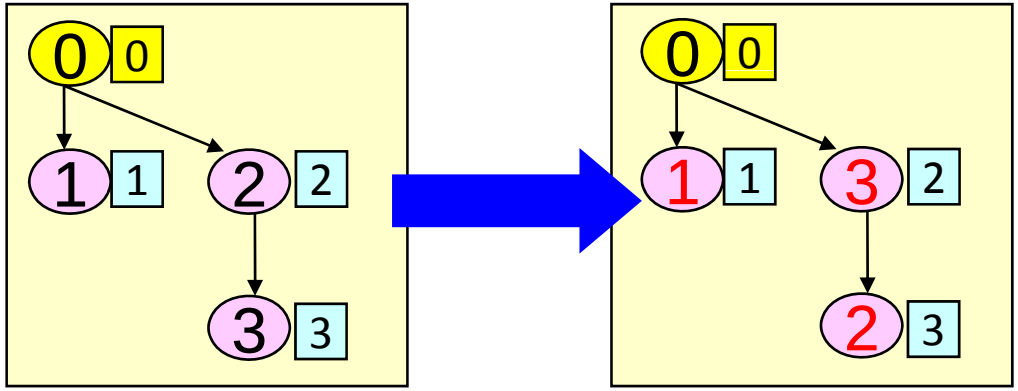
負荷の状況に応じた通信順序調整

- 全体通信中の待ち時間を計測



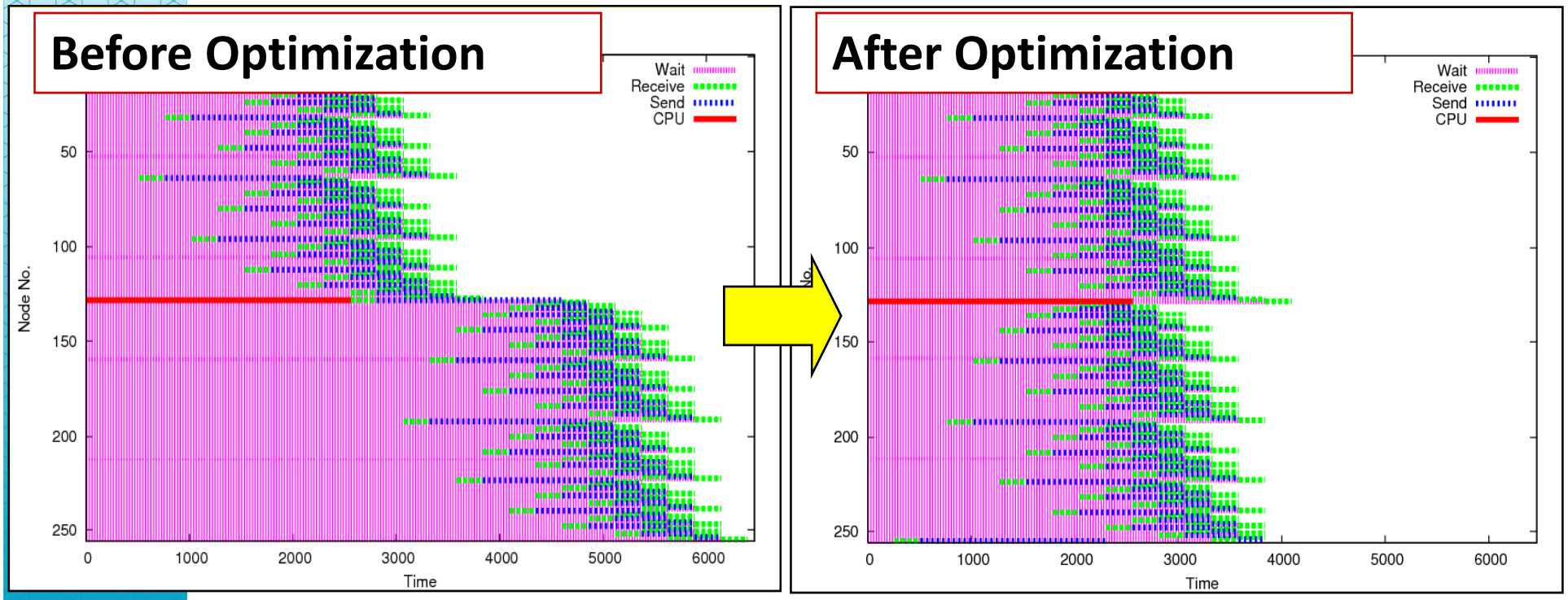
待ち時間短　⇒　高負荷

- 負荷状況に応じて全体通信アルゴリズム中の各ランクの位置を調整（＝順序調整）



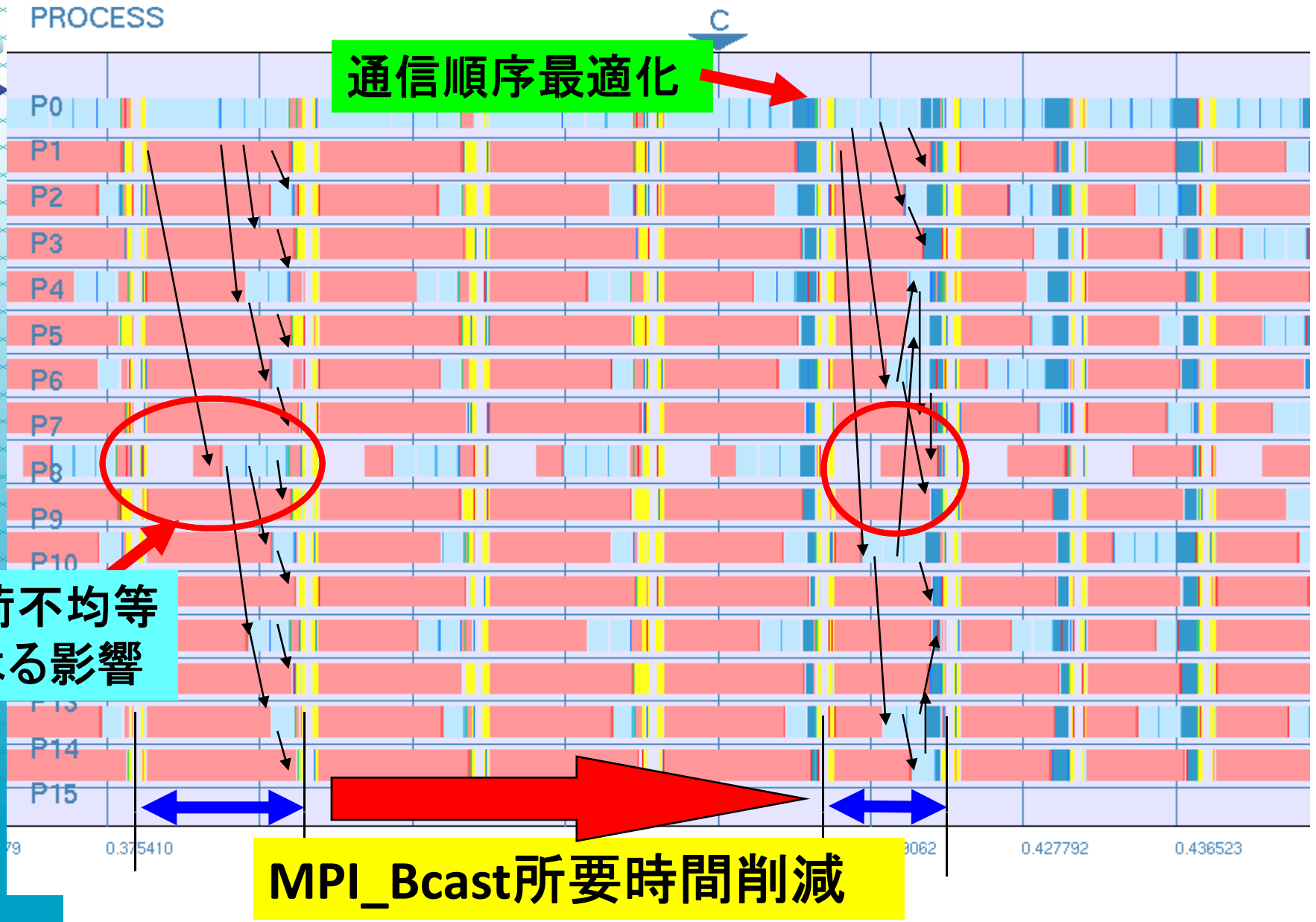
シミュレーションによる効果の予測

条件:
- ランク 128 のみに負荷(全体 256ランクの MPI_Bcast)
- 通信衝突等を考慮しない簡単な通信モデルを想定
- 通信最適化に要するコストを無視



- MPI_Bcast所要時間: -36%
- 全体の待ち時間: -31%

実際の効果

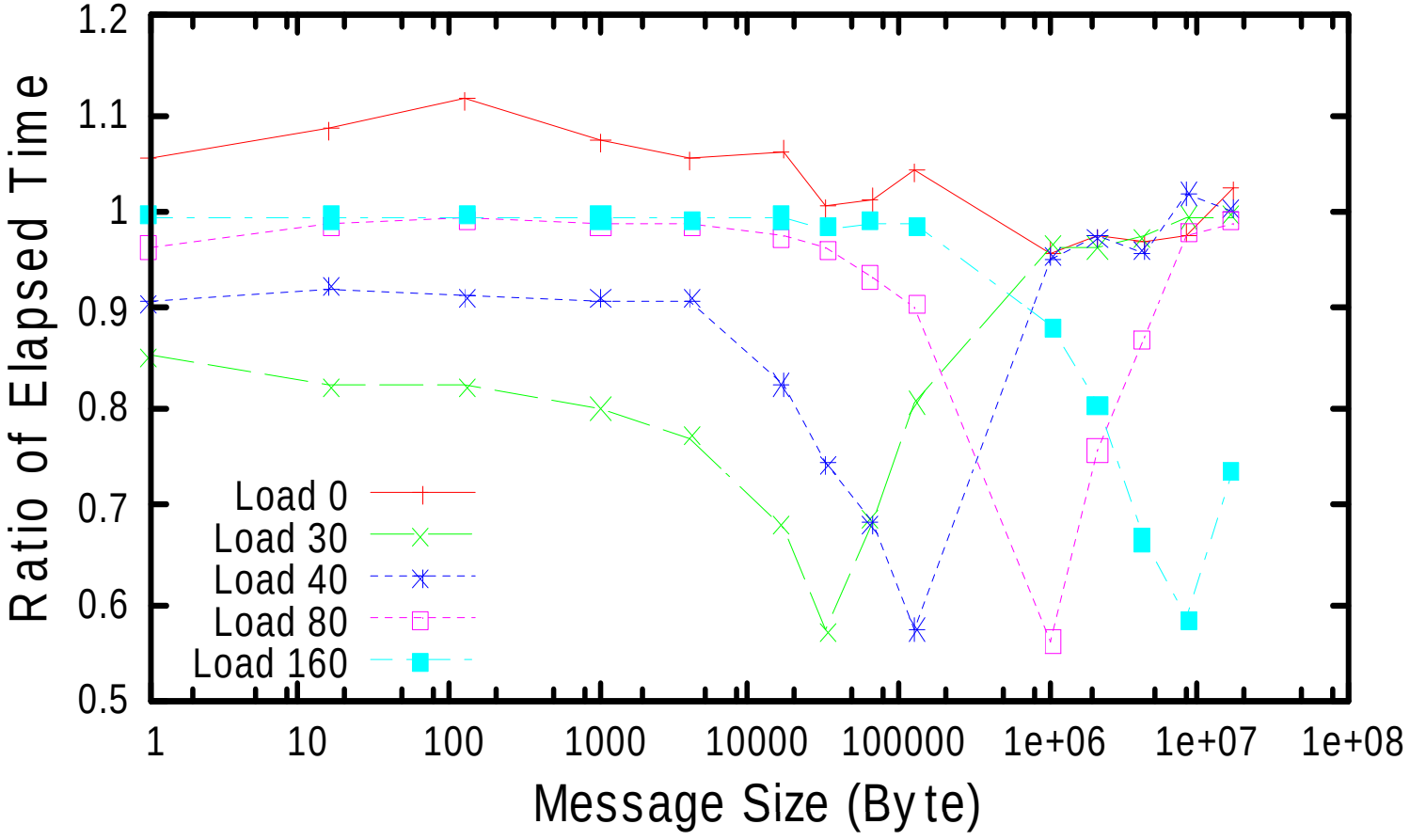

PROCESS
C
通信順序最適化
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P14
P15
負荷不均等
による影響
0.375410
0.427792
0.436523
MPI_Bcast所要時間削減

計測

- プログラム１：　擬似負荷プログラム
  - 中間位置にあるランクのみ
    MPI_Bcastの前で行列積の計算を実行
  - 最大効果の確認
- プログラム２：　疎行列積
  - CCS(Compressed Column Storage)で圧縮格納された疎行列同士の積
  - ランク0：　　行を抽出して MPI_Bcast
    それ以外：　　行を受け取ってベクトル行列積計算
- 環境：
  RSCC (Riken Super Combined Cluster)
  ＠理化学研究所

擬似負荷プログラム

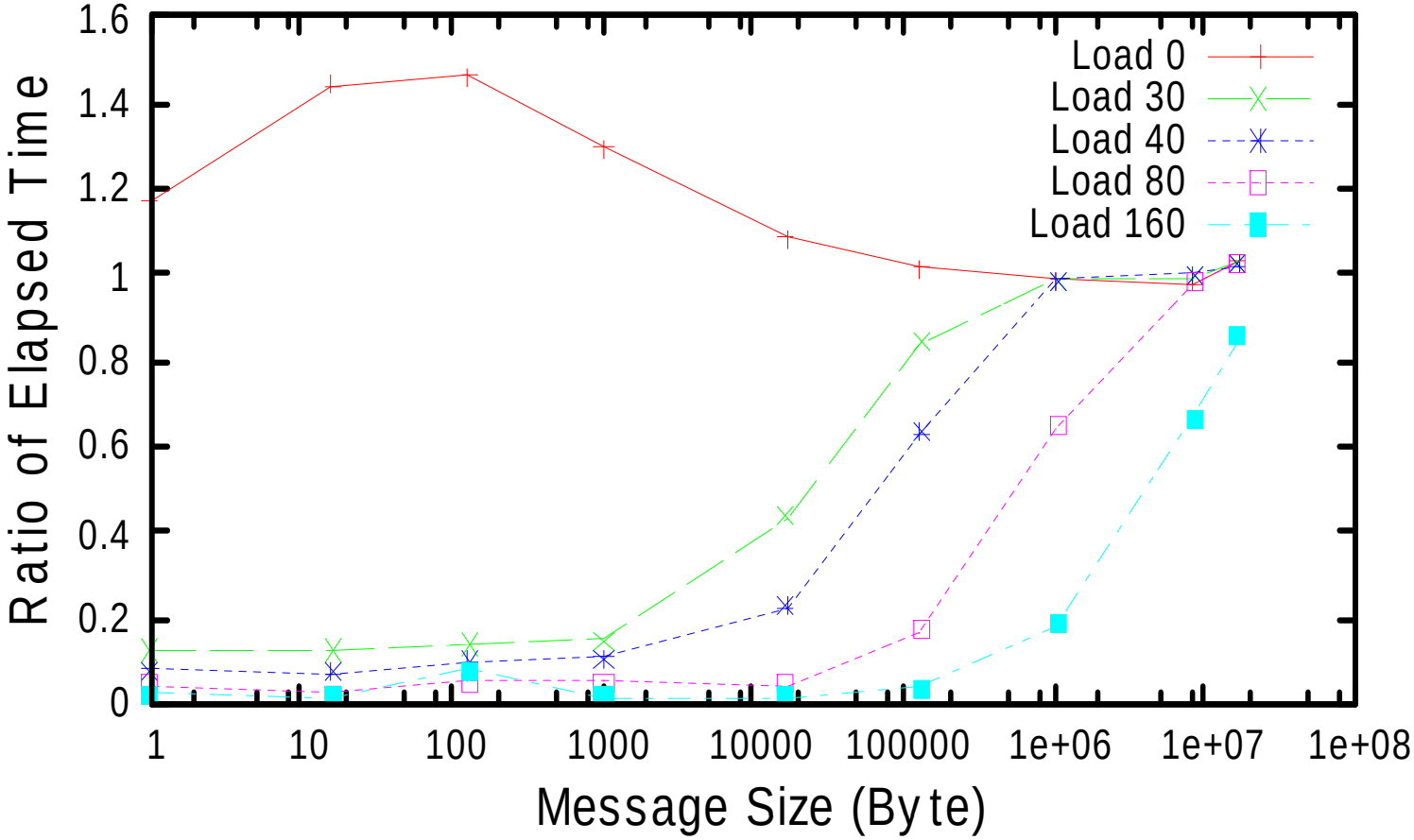
Ratio = MPI_Bcast所要時間の比
（最適化適用 / 最適化なし）



負荷バランスの度合いとメッセージサイズにより最適化効果が変動

MPI_Bcastの総内部通信時間

Ratio = 最適化適用 / 最適化なし

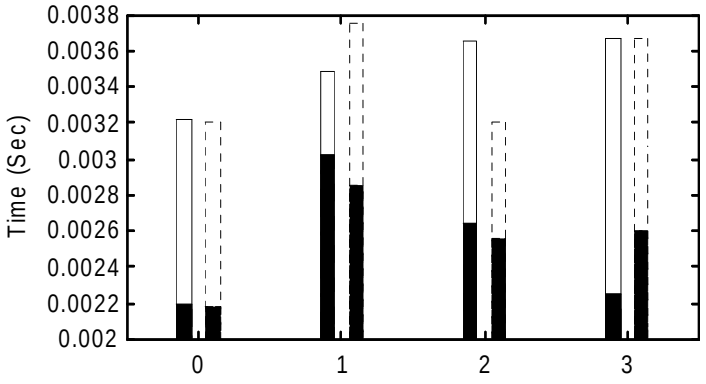


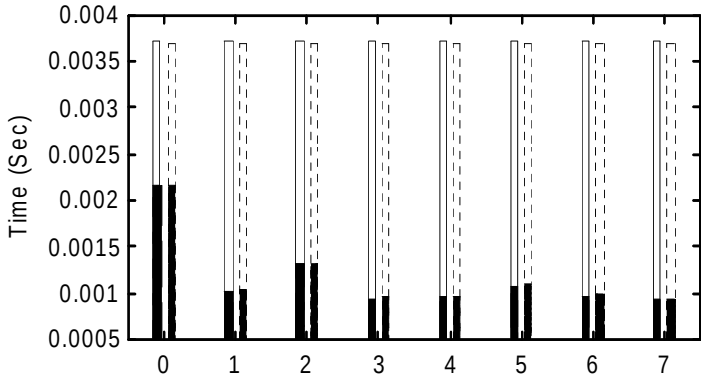
全ランクの合計待ち時間を大幅に削減

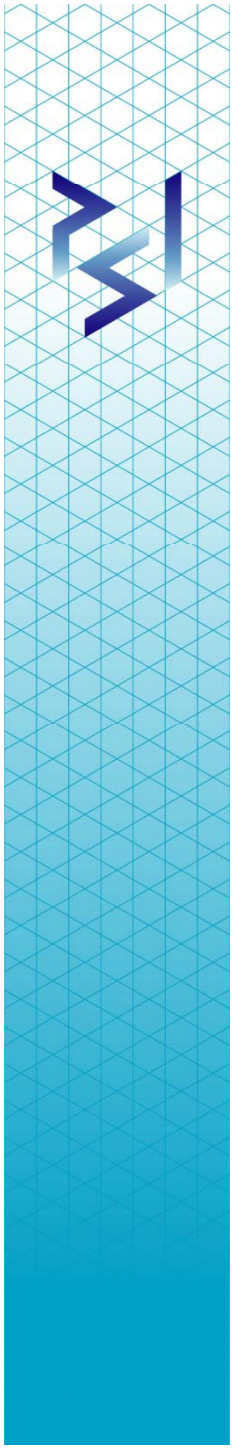
疎行列積

疎行列データ:BCSSTK32 (44609x44609)
from Matrix Market

		Workload Time			Broadcast Time			Total Time		
Type		Orig	Opt	Opt/Orig	Orig	Opt	Opt/Orig	Orig	Opt	Opt/Orig
Number of Process	4	2530	2550	1.008	977	909	0.930	3510	3460	0.986
	8	783	784	1.001	1690	1680	0.994	2470	2460	0.996
	16	350	349	0.997	2090	2070	0.990	2440	2420	0.992
	32	169	171	1.012	2370	2380	1.004	2540	2550	1.004
	64	82.7	82.7	1.000	2670	2670	1.000	2750	2750	1.000
	128	41.9	41.8	0.998	2890	2880	0.997	2930	2920	0.997







課題

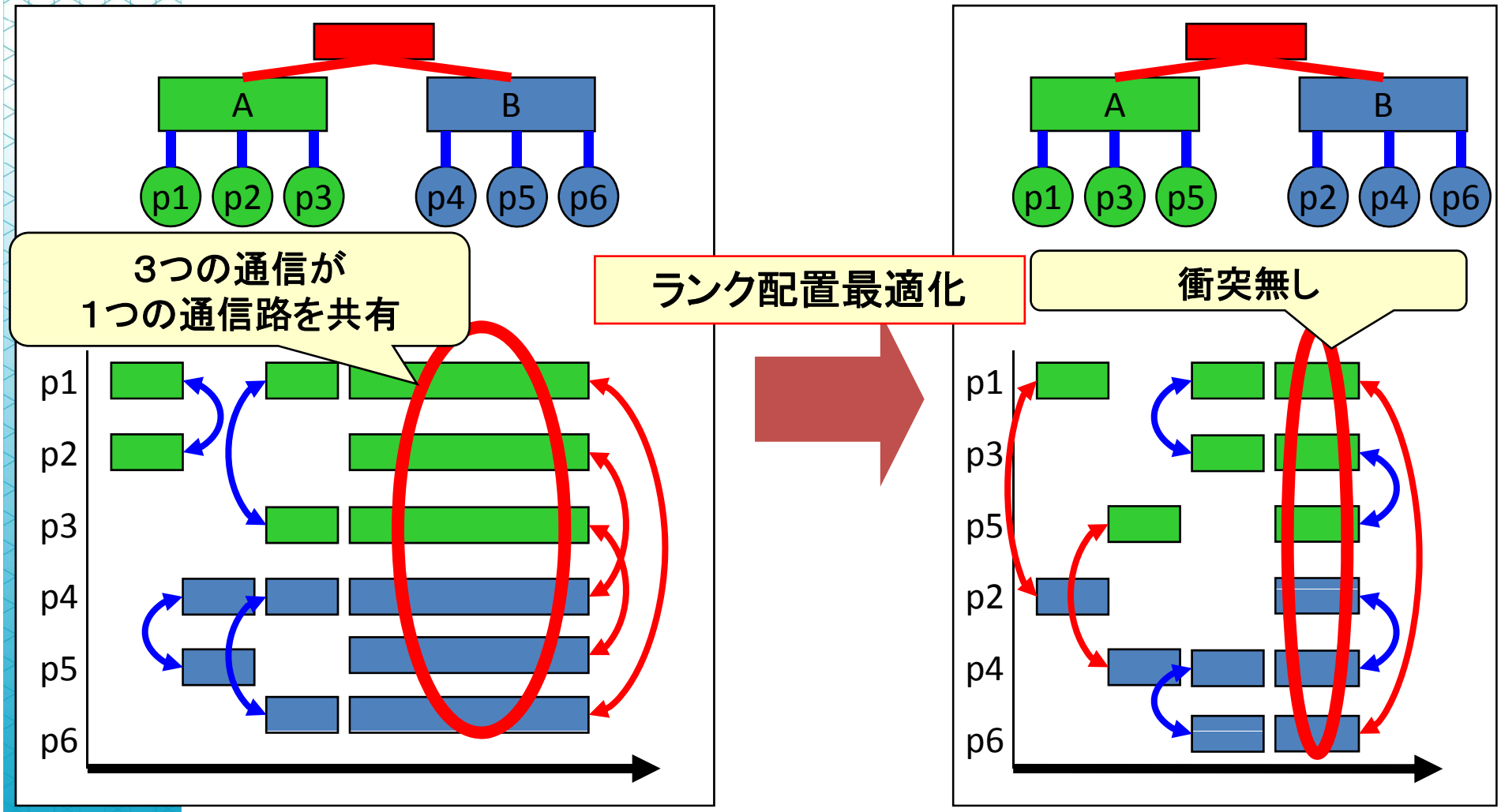
- 最適化コストによる性能低下

- To do:
  1. 最適化コスト削減
     - 特に負荷情報収集手段の改良
     - 状況に応じた選択的な適用

  2. プログラム中のMPI_Bcastの場所に応じた最適化

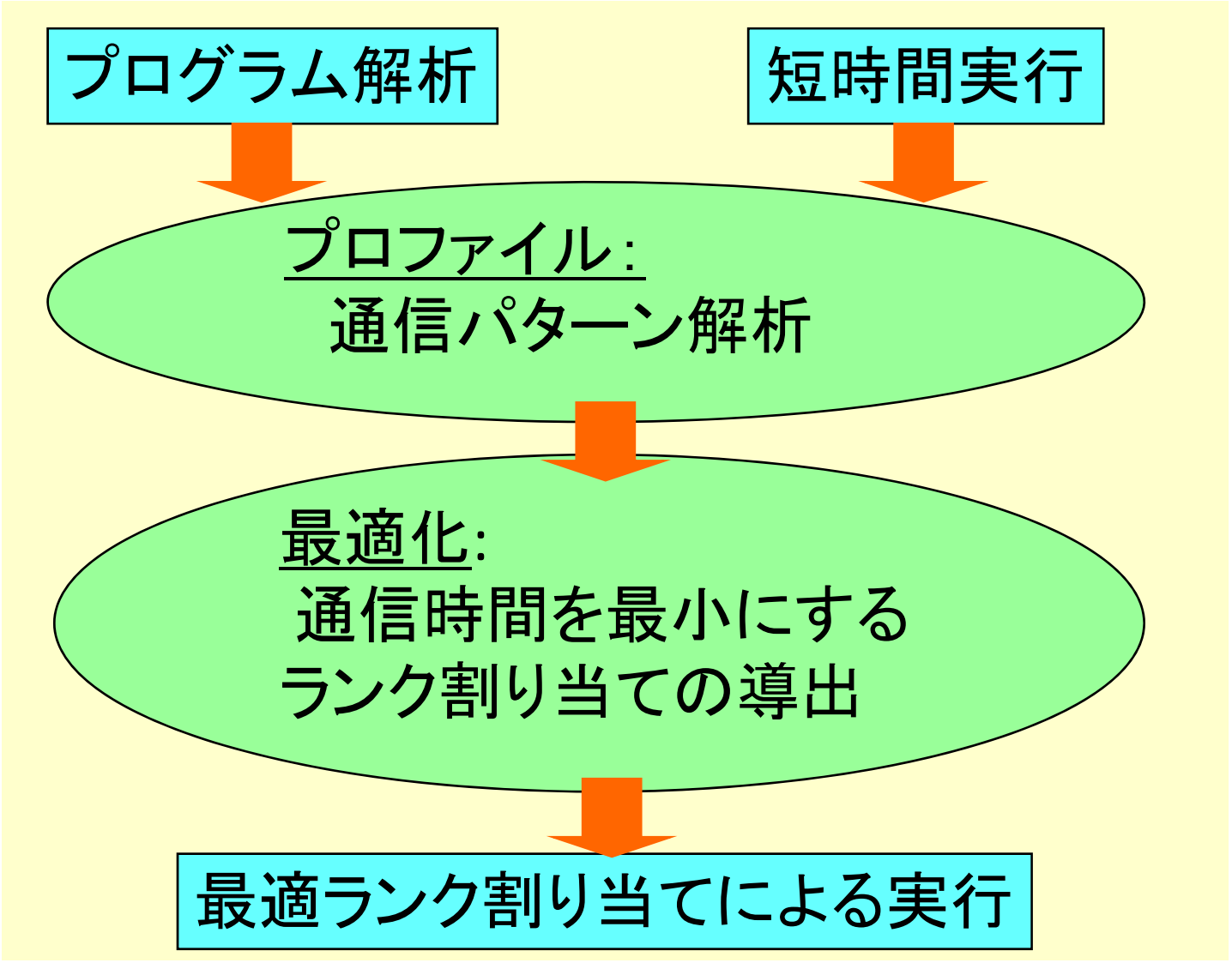
  3. 他の全体通信 (MPI_Scatter, MPI_Gather, MPI_Reduce等) への適用

ランク配置最適化



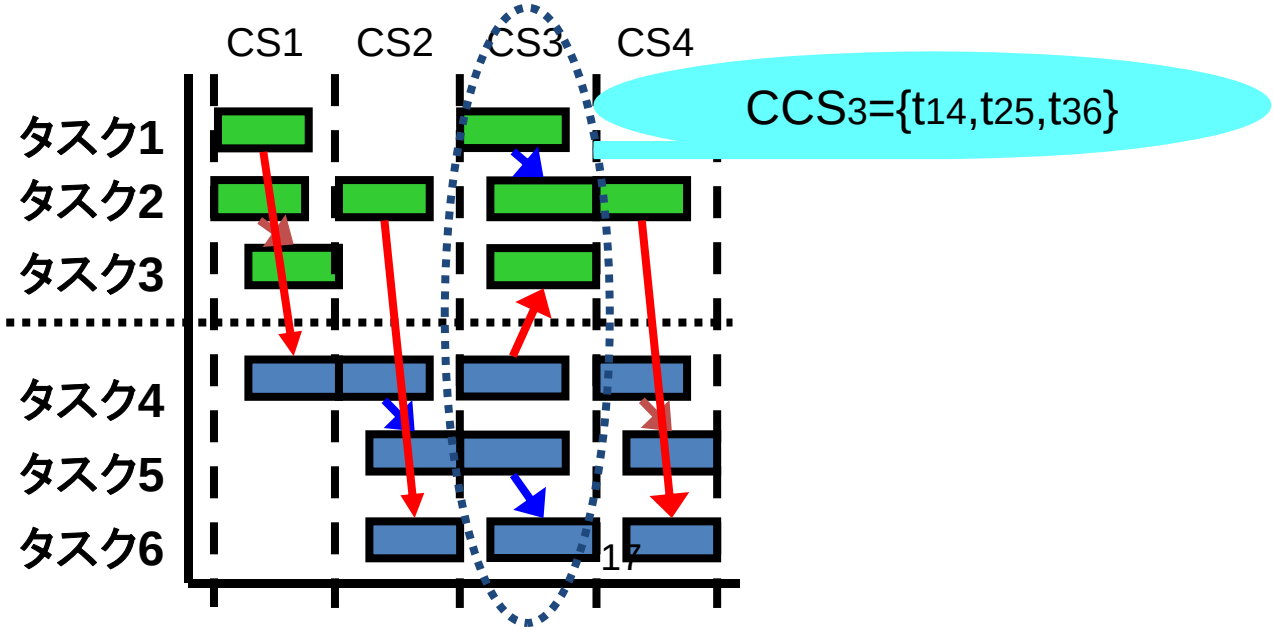
どちらも、合計通信距離は同じ
⇒ 既存の最適化技術では回避困難

ランク配置最適化の流れ



通信パターンの解析

- 同時に実行可能な通信の集合 CCS_iの抽出
  - CCS (Concurrent Communication Set)
  - i ： 通信ステップ（Communication Step)



最適化： 見積もり通信時間を最小にするランク配置

遅延　メッセージサイズ　衝突回数　通信帯域幅

$$\sum_{t=0}^{N-1} \max_{i=0,\cdots,n-1} (L(\pi(i),\pi(tork(i))) + S(t,i,tork(i)) \times coll(t,\pi(i),\pi(tork(i)))/B(\pi(i),\pi(tork(i))))$$

t　通信ステップ

N　通信ステップの総数

i　タスク番号

n　タスク総数

$\pi(i)$ タスクが割り付けられている計算ノードを返す関数

$tork(i)$　タスク i の通信先のタスクを返す関数

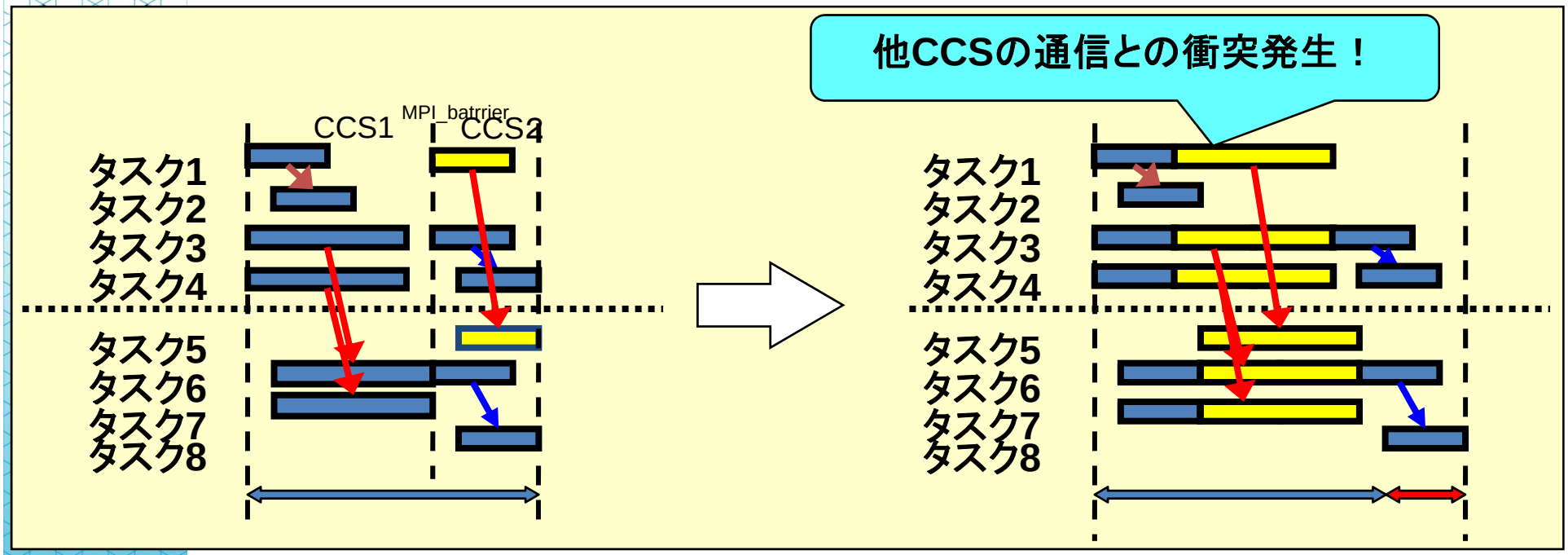
$L(p,q)$　計算ノード p，q 間の通信遅延を返す関数

$B(p,q)$　計算ノード p，q 間の通信帯域幅を返す関数

$S(t,i,j)$　通信ステップ t におけるタスク i からタスク j へのメッセージサイズを返す関数

$coll(t,p,q)$　通信ステップ t における計算ノード p，q 間の各経路で発生した通信の衝突回数の最大値+1 を返す関数

同期関数によるCCSの分離



- 複数のCCSが同時に実行
  ⇒ **予期しない衝突の発生**
- 同期関数（MPI_Barrier）により CCS を分離

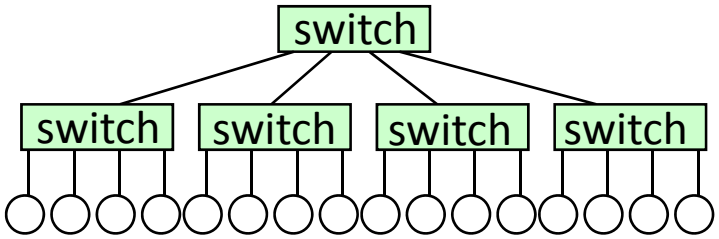
```
MPI_Barrier(comm)
if(t %2 == 0)
 MPI_Send( smsg, t+1 )
el se
 MPI_Recv( rmsg,  t-1)

MPI_Barrier(comm)
if(t %2== 0)
 MPI_Recv( smsg1, t+1)
else
 MPI_Send( rsmg1, t-1 )
```

実験結果

■使用プログラム
- Recursive doubling アルゴリズムによる全対全通信
- NAS Parallel Benchmark 中のCG
- ASCI Purple Benchmark 中のUMT2000

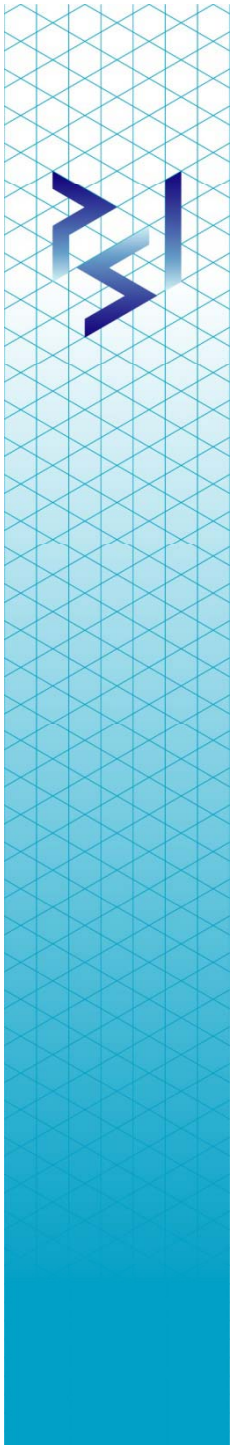
■実験環境
階層的なツリー構造の Gigabit Ethernet で接続されたPCクラスタ
(Dual Xeon 3.0GHz x 16ノード)



■通信時間の比較

	Recursive Doubling	CG	UMT2000
最適化無し	53.6	31.2	109.8
通信距離による最適化	53.6	31.2	90.6
衝突を考慮した最適化	**43.2**	**24.3**	**60.1**

(msec)



課題

- 同期関数コストによる性能低下
  - メッセージサイズが小さい場合
    （＝最適化による性能向上幅が小さい場合）

⇒ 同期関数挿入箇所の削減

- 最適配置の探索時間
  - 現在は全探索　⇒　O(P!) (P = プロセッサ数)

⇒ 発見的手法による探索