Optimization 2: Communication Optimization

Osamu Tatebe

tatebe@cs.tsukuba.ac.jp

Center for Computational Sciences,

University of Tsukuba

Agenda

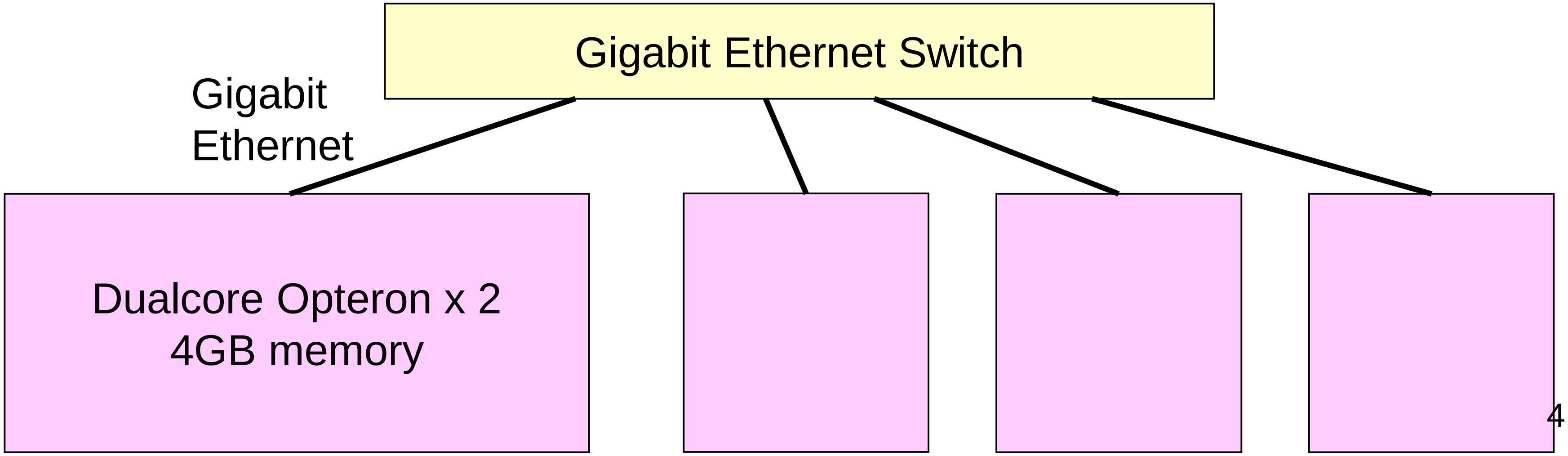
- Basic communication performance
  - Point-to-point communication
  - Collective communication
- Profiling
- Communication optimization technique
  - Communication reduction
  - Communication latency hiding
  - Communication blocking
  - Load balancing

Basic Performance

- Performance for basic communications **should be understood** to optimize communication
  - Understand performance in various communication patterns
  - Decide the block size of communication blocking
  - Improve the performance communication library compared with the peak network performance

PC Cluster Platform [P1]

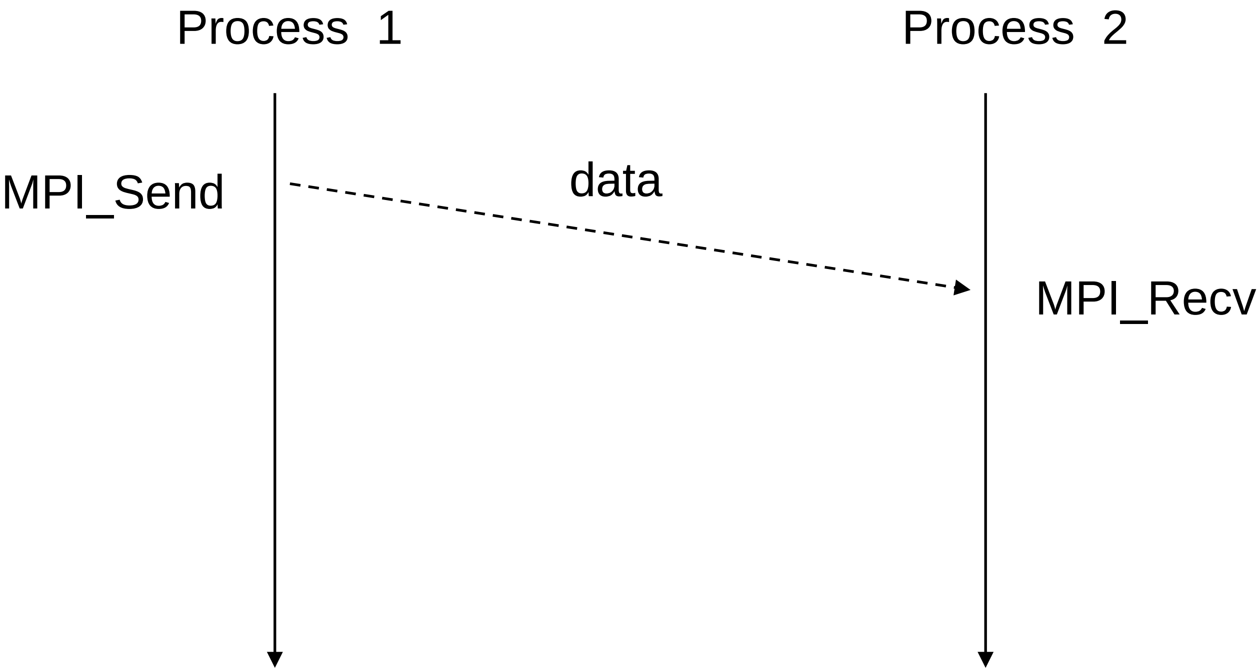
- 4 cluster nodes
  - 2.6GHz Dualcore Opteron x 2 sockets (4 cores)
  - 4GB memory
  - Linux 2.6.18-1.2798.fc6
  - OpenMPI 1.1-7.fc6
- Connected by Gigabit Ethernet
  - Theoretical peak in TCP is 949 Mbps (= 113.1 MB/sec)



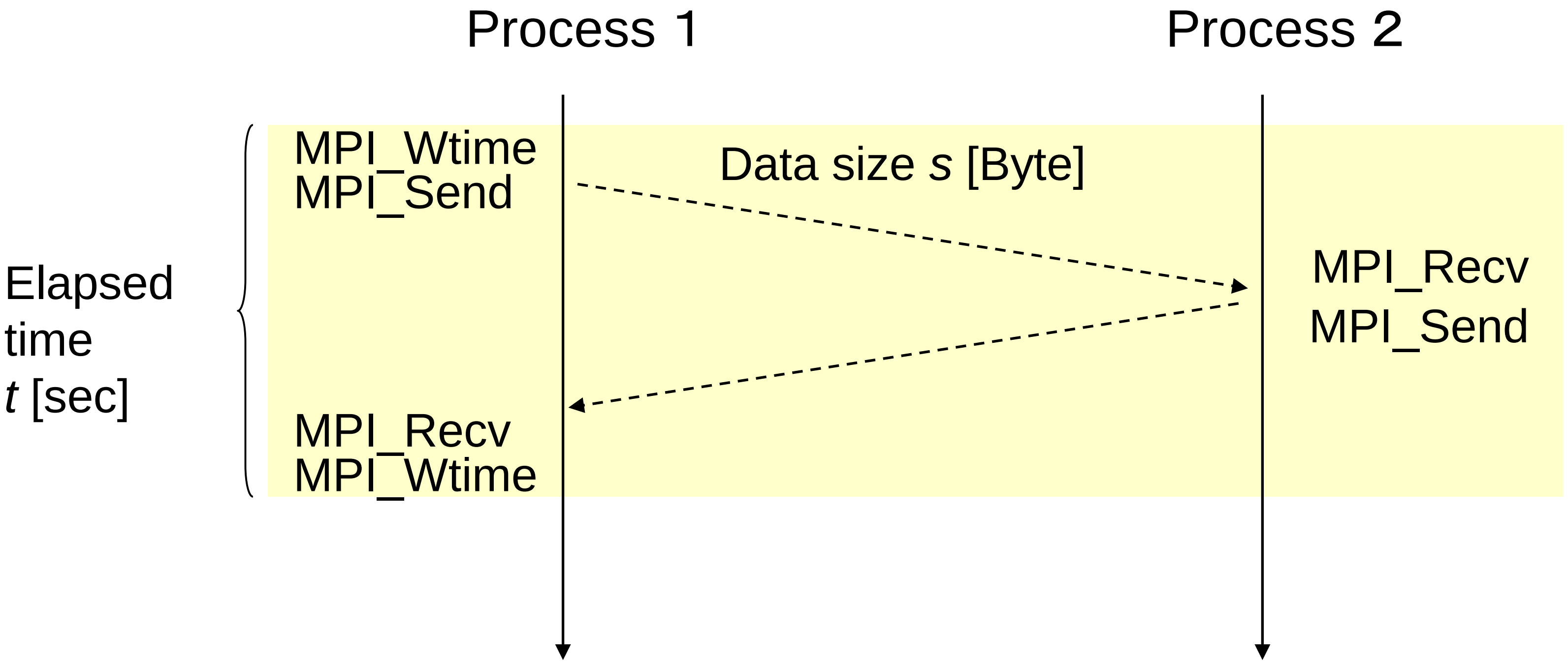
Supercomputer [P2]

- Oakforest-PACS 4 nodes
  - 1.4GHz Xeon Phi (Knights Landing; KNL) (68 cores)
  - 96GB DDR4 + 16GB MCDRAM
  - Intel MPI
- Connected by Omni-Path
  - Peak bandwidth is 100 Gbps
- No memory location optimization

Performance of point-to-point communication



PingPong Benchmark (1)

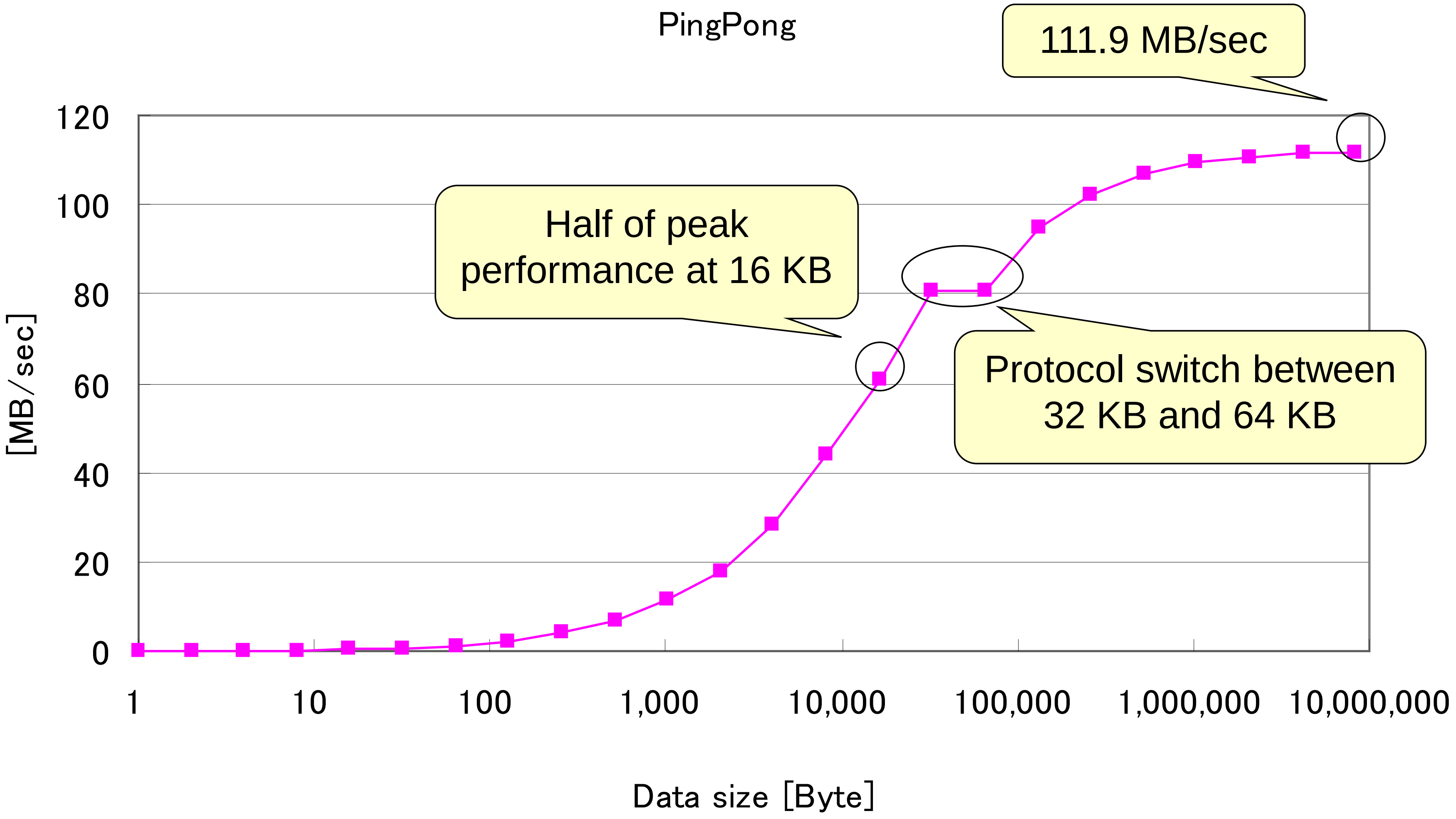


Network bandwidth $s/(t/2)$ [Byte/sec]

PingPong Benchmark (2)

```
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); // size, time, bandwidth
}
```

[P1] PingPong Benchmark



111.9 MB/sec

Half of peak performance at 16 KB

Protocol switch between 32 KB and 64 KB

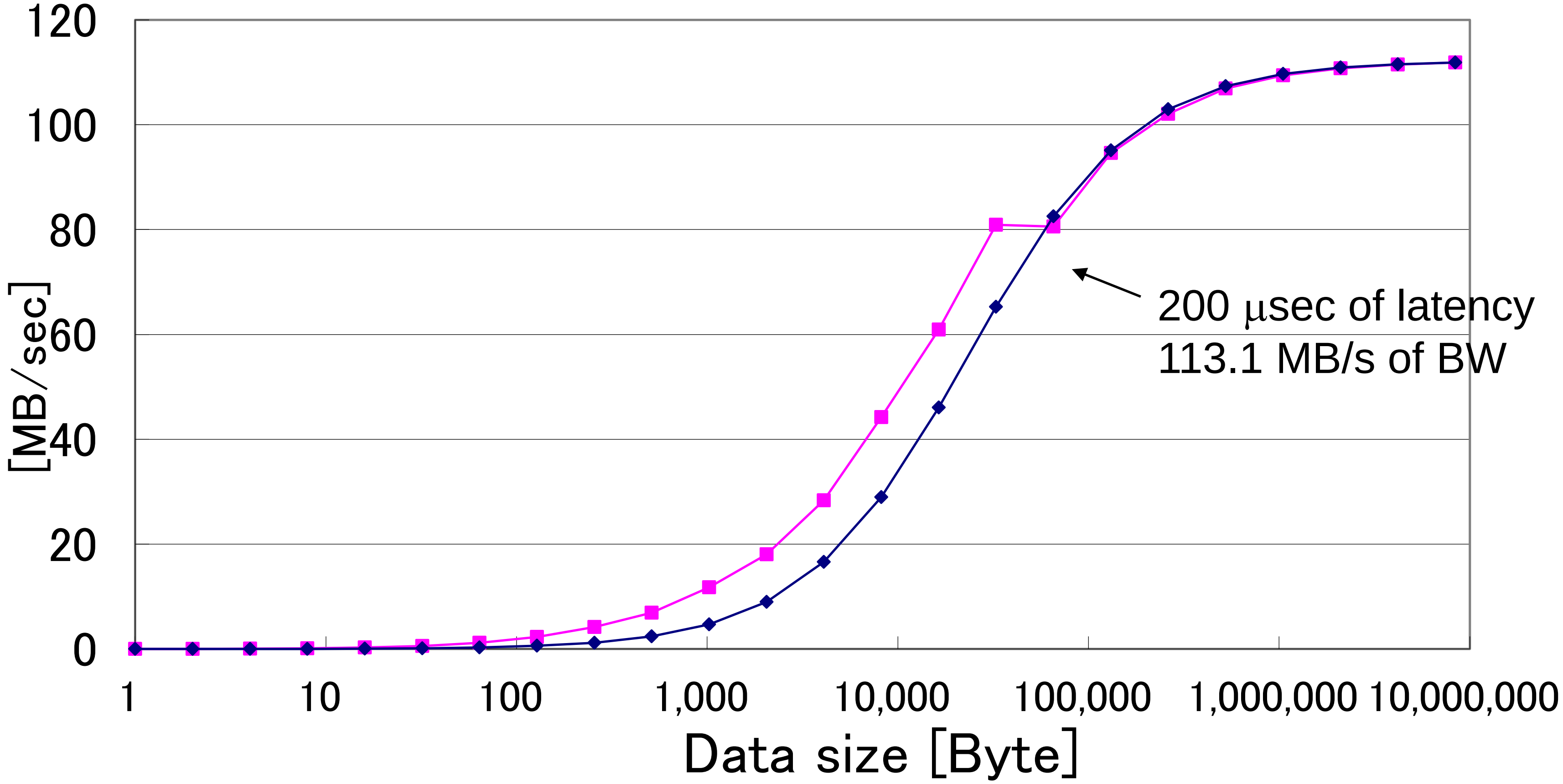
Protocol of point-to-point communication

- Eager protocol (1-way protocol)
  - for relatively small size of messages
  - A sender sends both the message header and the message body (data, payload) at the same time
  - It can reduce the communication latency, but incurs copy overhead at the receiver
- Rendezvous protocol (3-way protocol)
  - for larger size of message
  - A sender sends the message header, and waits for the acknowledgement
  - The sender sends the message body
  - It can achieve good communication bandwidth by reducing the copy overhead, but has longer latency than the eager protocol

Protocol of point-to-point communication (continued)

- MPI selects one of several protocols according to the message size
- It is visible if we carefully measure the performance with various message size
- Most MPI allows for users to specify the threshold of the message size for the protocol switch to optimize the communication performance

[P1] Comparison with theoretical curve



200 μsec of latency
113.1 MB/s of BW

Theoretical curve $s/\left(L+s/B\right)$

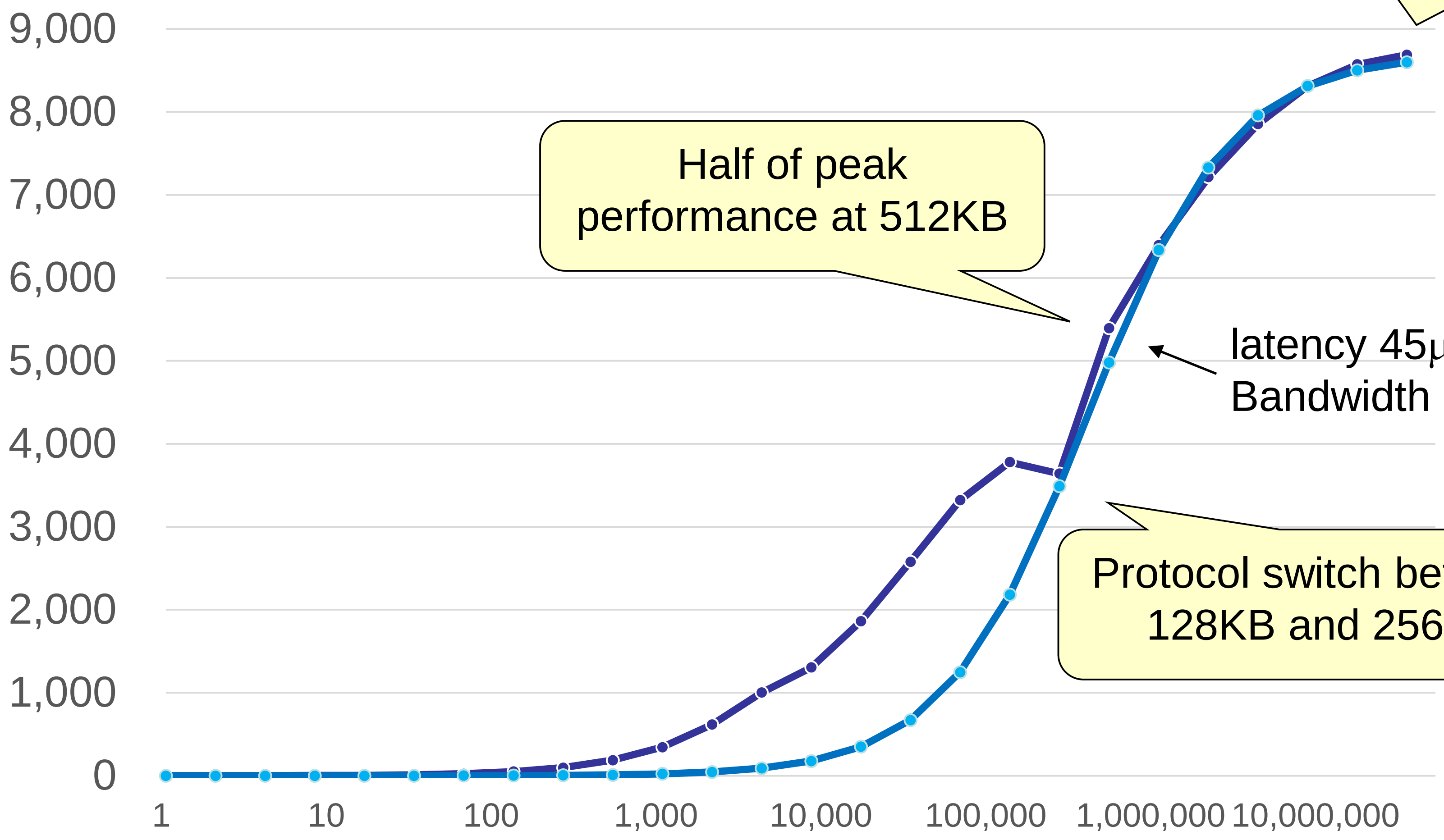
$N_{half}=BL$ L latency B bandwidth

[P1] PingPong Benchmark Summary

- Larger data size gets better performance
- Cf. theoretical peak is 113.1 MB/sec
- More than half → 16 KB or larger
- More than 90% of peak → 512 KB or larger

- Performance follows the curve of 200μsec latency in long message
  - Although latency of 1-byte PingPong is 563 μsec

[P2] PingPong Benchmark



8.7 GB/sec

Half of peak performance at 512KB

latency 45µsec
Bandwidth 8.7GB/s

Protocol switch between 128KB and 256KB

[P2] PingPong Benchmark Summary

- More than half→512KB or larger
- Performance follows the curve of 45μsec latency in long message
  - – Although latency of 1-byte PingPong is 2 μsec

Intel® MPI Benchmark

- Basic MPI Benchmark Kernel
- MPI1
  - PingPong
  - PingPing

Single Transfer

  - Sendrecv
  - Exchange*

Parallel Transfer

  - Bcast
  - Allgather
  - Allgatherv
  - Alltoall*
  - Alltoallv*
  - Reduce
  - Reduce_scatter
  - Allreduce*
  - Barrier

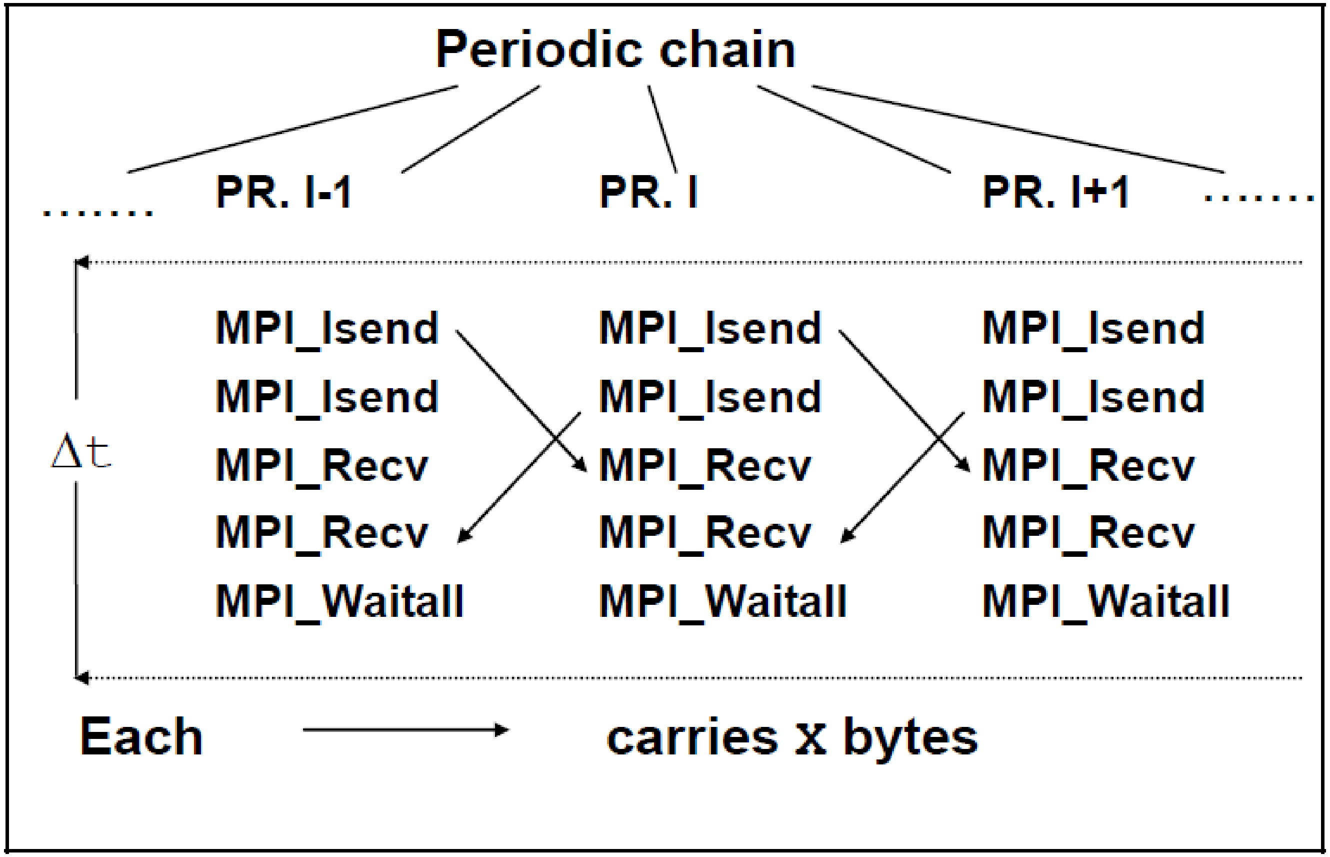
Collective

  - Multiple version that executes above in parallel

- 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 Pattern

- Communication pattern to exchange border elements



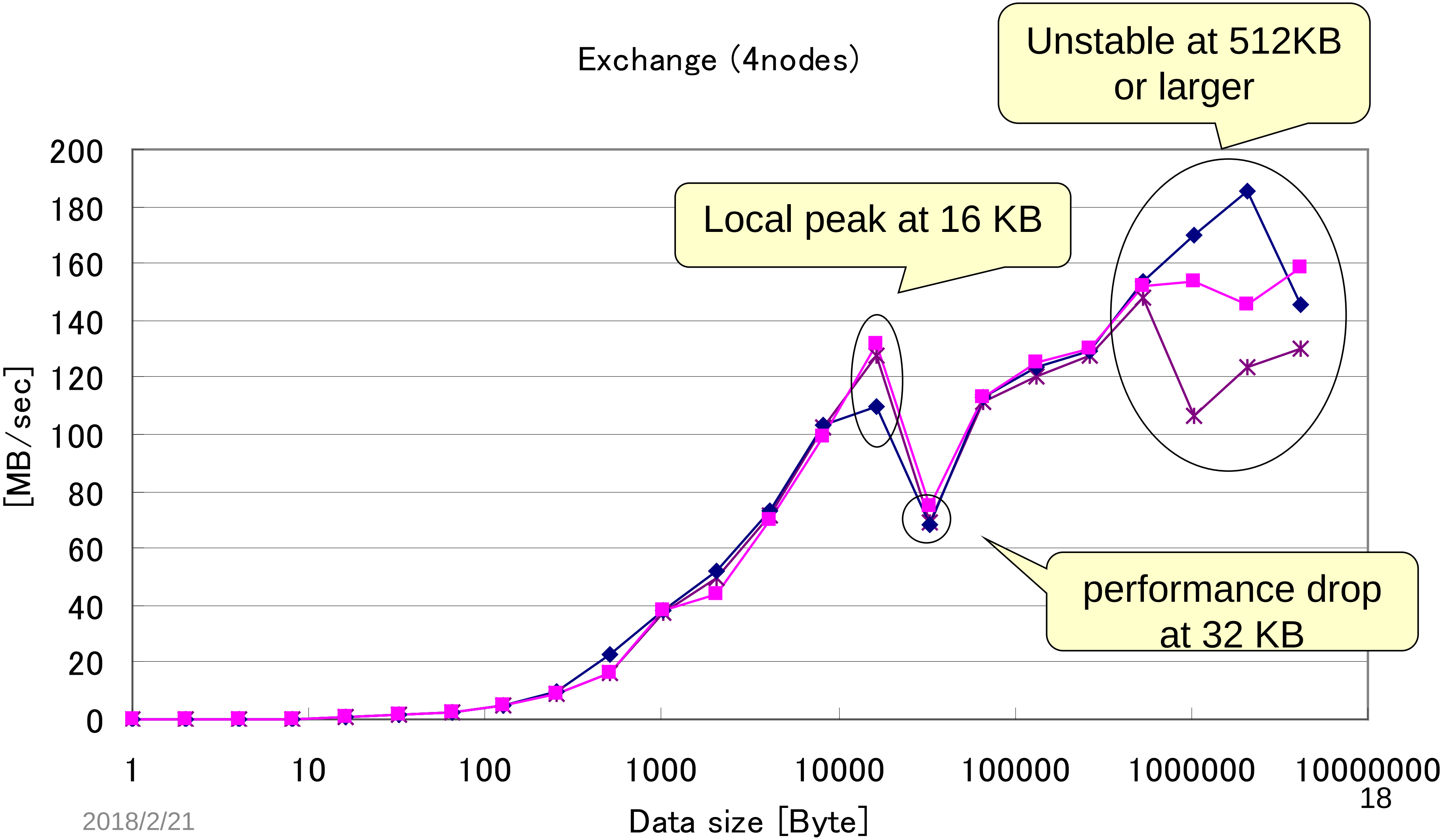
Periodic chain

…….	PR. I-1	PR. I	PR. I+1	…….
Δt	MPI_Isend	MPI_Isend	MPI_Isend	
	MPI_Isend	MPI_Isend	MPI_Isend	
	MPI_Recv	MPI_Recv	MPI_Recv	
	MPI_Recv	MPI_Recv	MPI_Recv	
	MPI_Waitall	MPI_Waitall	MPI_Waitall	

Each → carries x bytes

*From Intel MPI Benchmarks Users Guide and Methodology Description

[P1] Exchange (4 nodes) [3 trials]



Unstable at 512KB or larger

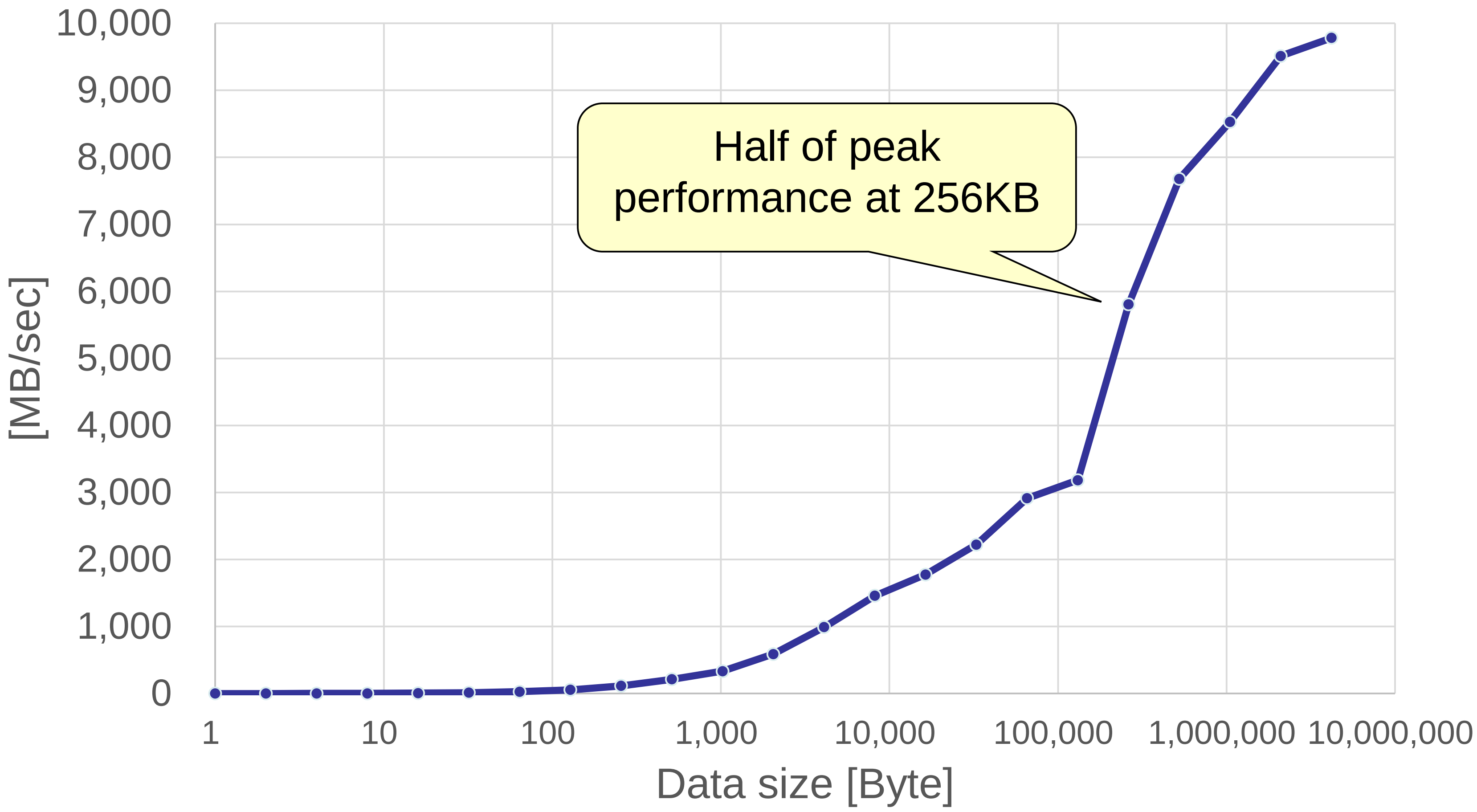
Local peak at 16 KB

performance drop at 32 KB

[P1] Exchange (4 nodes) Summary

- Basically larger data size gets better performance except around 32 KB
- Cf. Theoretical peak is 2*113.1 = 226.2 MB/sec
- More than half → 16KB and 128 KB or larger
  - Less than half at 32 KB and 64 KB
- Unstable at 512 KB or larger due to packet loss and RTO

[P2] Exchange (4 nodes)



Half of peak performance at 256KB

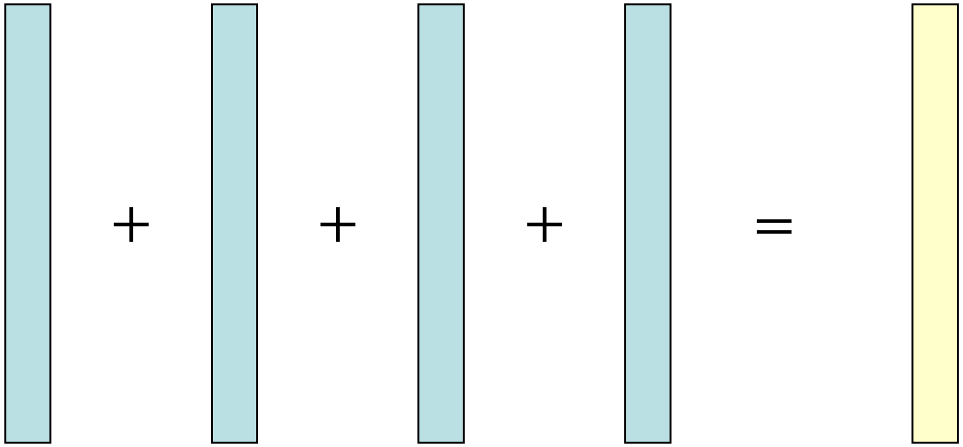
[P2] Exchange Summary

- Larger data size gets better performance
- More than half of peak performance when 256KB or larger
- Performance is stable
  - Omni-Path does not drop packets

Allreduce

- Do specified operation (sum, max, logical and/or, …) among arrays of each process, and store the result in all processes
- Example of MPI_SUM

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



Array of process 1

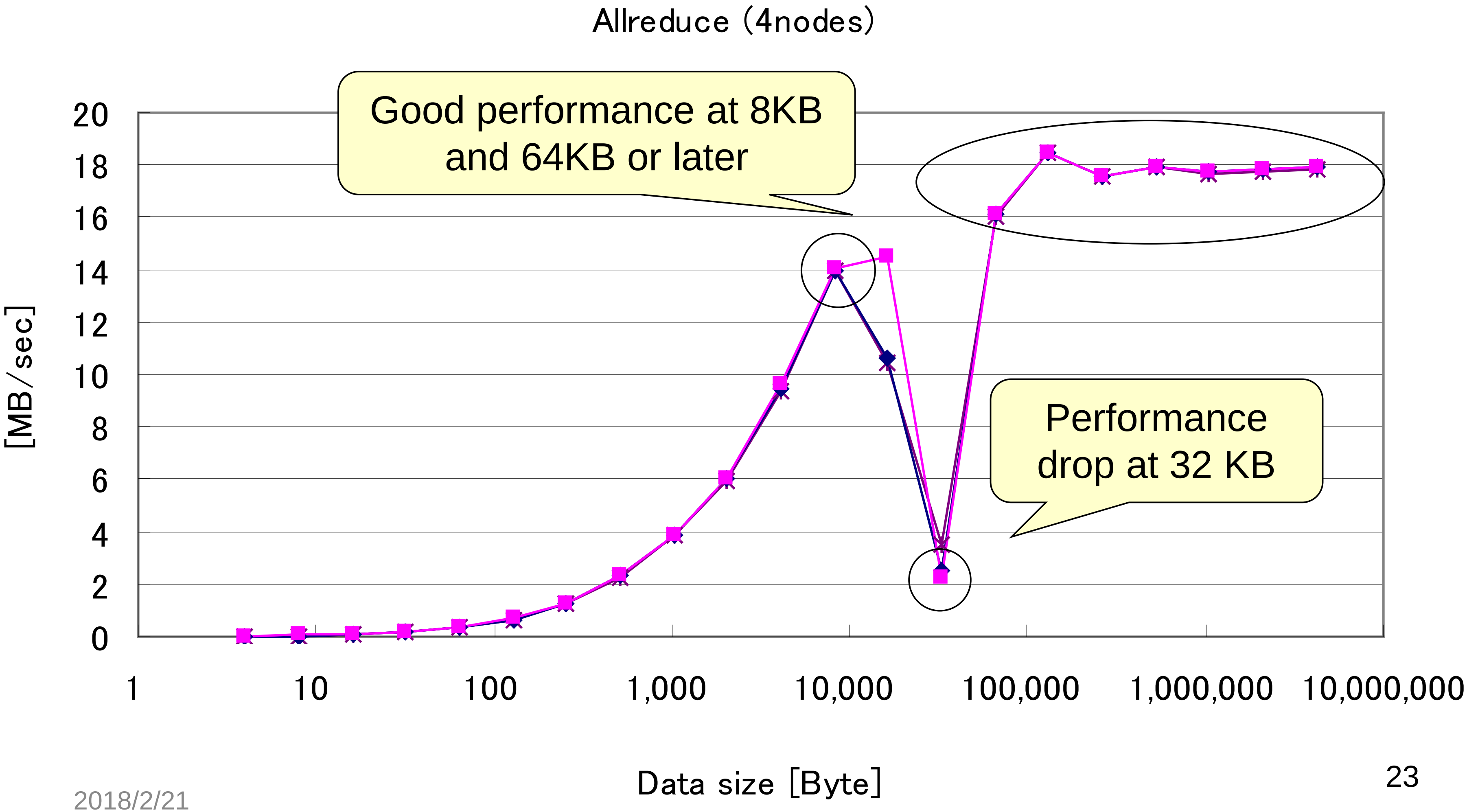
Array of process 2

Array of process 3

Array of process 4

All processes have the result

[P1] Allreduce (4 nodes)
[data size / time]



Good performance at 8KB and 64KB or later

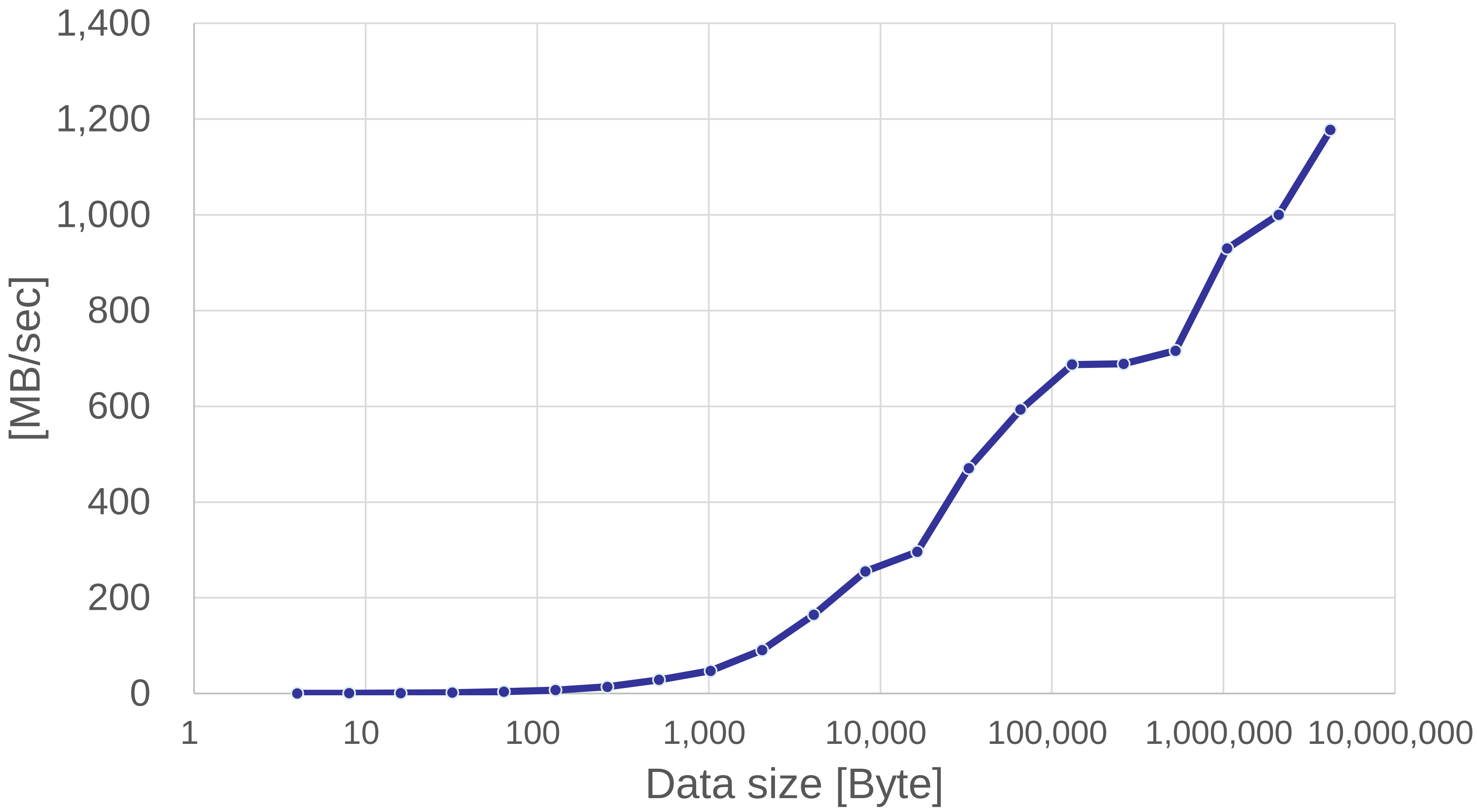
Performance drop at 32 KB

[P1] Allreduce Summary

- Basically larger data size gets better performance except around 32 KB
- Good performance is achieved at 8 KB and 64 KB or larger

[P2] Allreduce (4 nodes)

[data size / time]



[P2] Allreduce Summary

- Larger data size gets better performance
- Performance is stable
  - Omni-Path does not drop packets

Profiling

- Understand the behavior of programs
  - Frequently called functions
  - Time-consumed functions
  - Call tree
  - Memory usage of functions, …
- Understand the most time-consumed code
- Understand synchronization and load imbalance in parallel programs

Profiler is required not to change the behavior of parallel program so much

Communication profiling by users

- Users insert an instrumenting code at the point of interest by themself
- Put "wall clock measuring" (ex. MPI_Wtime, gettimeofday()) before and after to measure time of a certain block
  - for each MPI function
  - for some important blocks
- The accuracy of measuring "ticks" depends on the system

```
double t1, t;

t1 = MPI_Wtime();
MPI_Allgather(....);
t = MPI_Wtime() – t1;
```

- It is easy, but there are more sophisticated tools

tlog – time log

- Light-weight profiling library
  - 16 B of memory space for each event
- 9 kinds of single events and 9 kinds of interval events
  - It can be extended since event number field is 8 bit
- Record the elapsed time in seconds from tlog_initialize
  - Time difference among processes is measured in tlog_initialize
  - Recorded time is “absolute” time in parallel processes relative to tlog_initialize
- Temporal URL for download
  - http://www2.ccs.tsukuba.ac.jp/workshop/HPCseminar/2011/software/tlog-0.9.tar.gz

tlog – major API

void tlog_initialize(void)

initializes the tlog environment. It should be called after MPI_Init

void tlog_log(int event)

records a log of the specified event

void tlog_finalize(void)

outputs the logs to trace.log. It should be called before MPI_Finalize()

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

Example - cpi.c

- Test program that computes π

```
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);
/* compute mypi (partial sum) */
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) /* display the result */
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();
```

Example – compilation of cpi

- How to link tlog library

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

- How to install tlog library and tlogview

```
% ./configure
% make
% sudo make install
```

Example to install in /usr/local

Example – output of cpi

```
$ 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 0.0000083333333318
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 ...
```

measurement of time difference among nodes (output in debug mode)

output in debug mode

Output of program

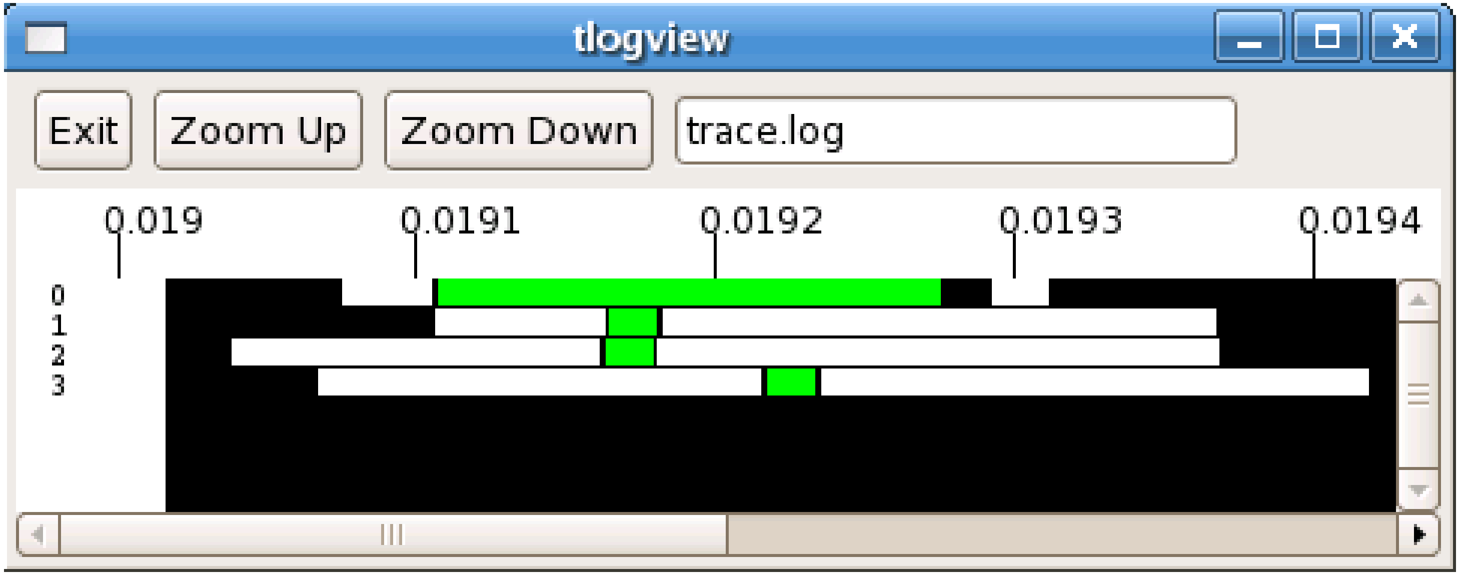
output in debug mode

Profiling result of cpi (1)

- tlogview – visualization tool for tlog output

```
% tlogview trace.log
```

- Profiling example when using 4 processes



trace.log

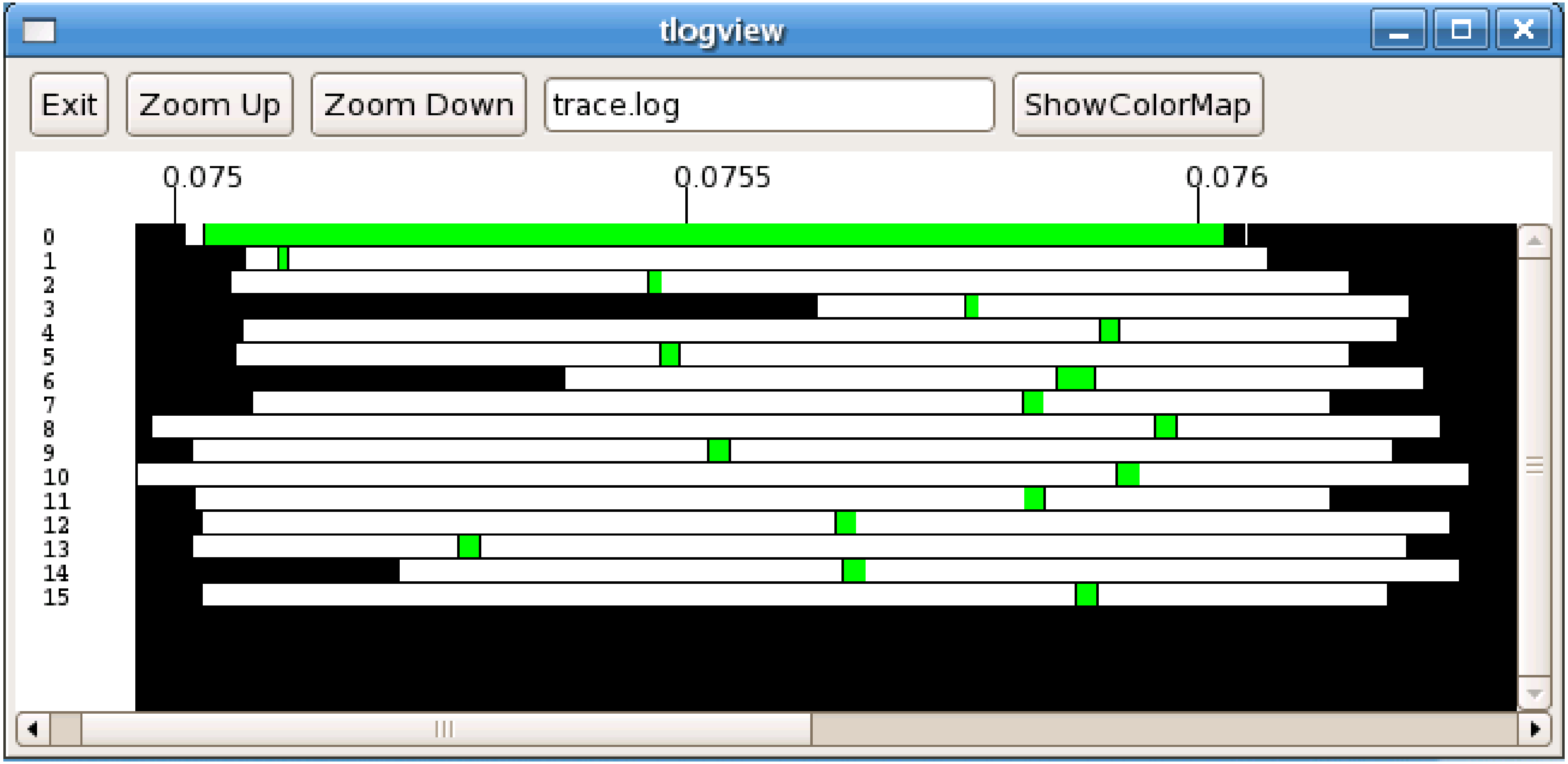
□ MPI_Bcast

🟩 MPI_Reduce

Elapsed time from tlog_initialize in seconds
(adjusted using the time difference among nodes)

Profiling result of cpi (2)

- Profile example when using 16 processes



□ MPI_Bcast 🟩 MPI_Reduce

Communication optimization

- Communication reduction*
- Load balancing*
- Communication blocking
  - Basically larger data size is better performance
- Communication latency hiding for short message communication
  - Overlapping computation and communication
  - Pipeline execution

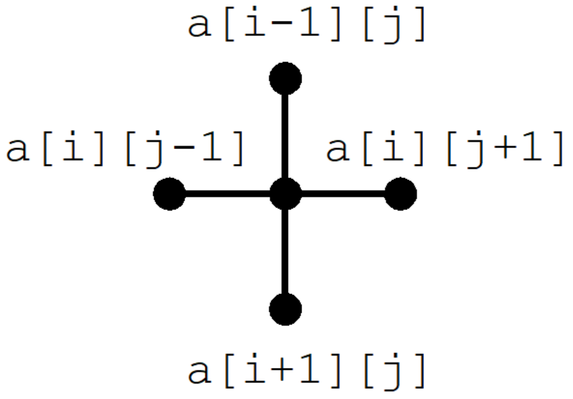
Communication blocking

- Data size is a major factor for communication performance
- Communication blocking enlarges the data size by aggregating the communication data
  - Block distribution of data
  - Aggregation of multiple iterations (temporal blocking)

Example of communication blocking – Jacobi method

- Solving a sparse matrix that arises when discretizing 2D Laplace equation in 5 point stencil

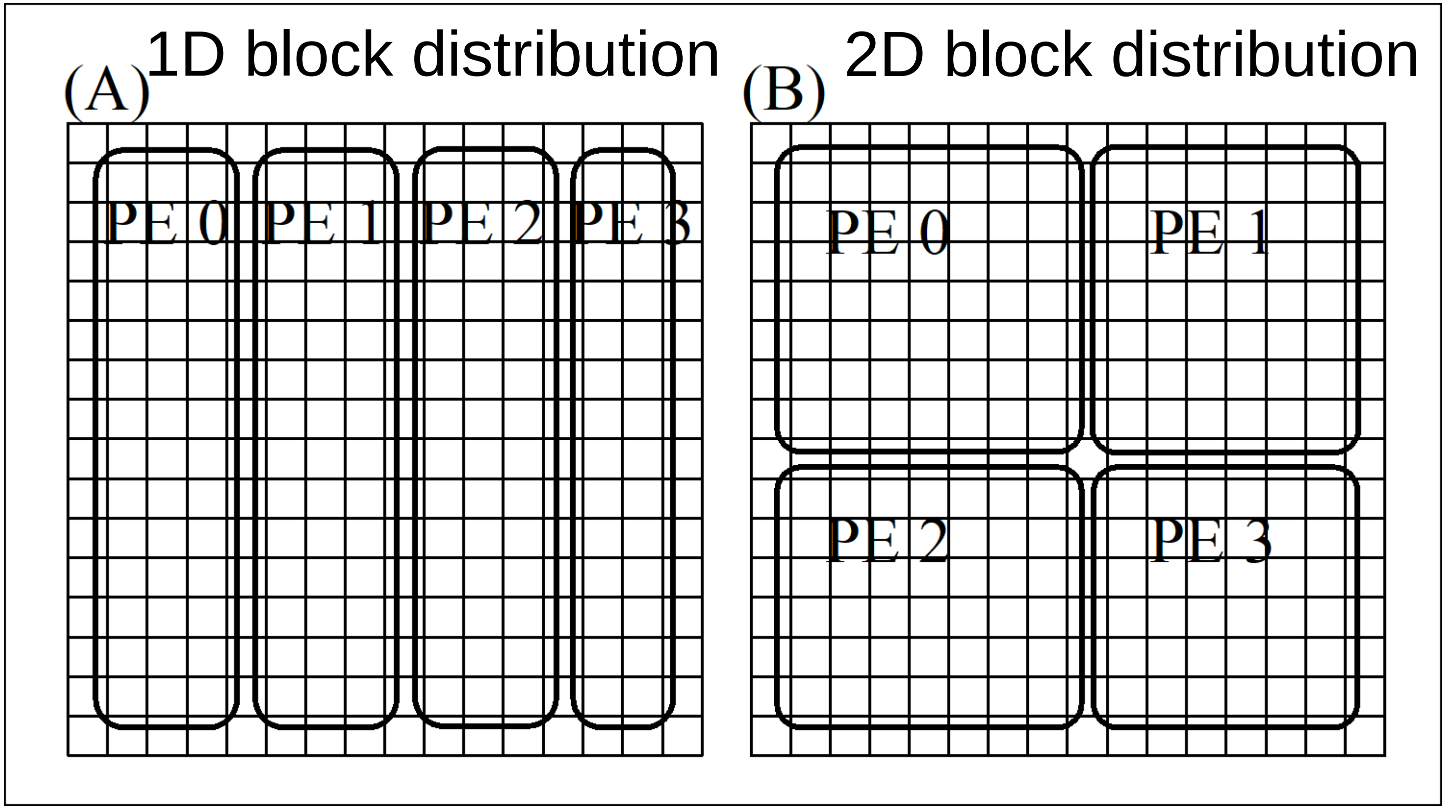
```
jacobi() {
  while (!converge) {
    for(i = 1; i < N - 1; ++i)
      for(j = 1; j < N - 1; ++j)
        b[i][j] = .25 *
              (a[i - 1][j] + a[i][j - 1]
               + a[i][j + 1] + a[i + 1][j]);
    /* convergence test */
    /* copy b to a */
  }
}
```



Data dependency

*In fact, not to use Jacobi method but RB-SOR etc.

Block distribution of data

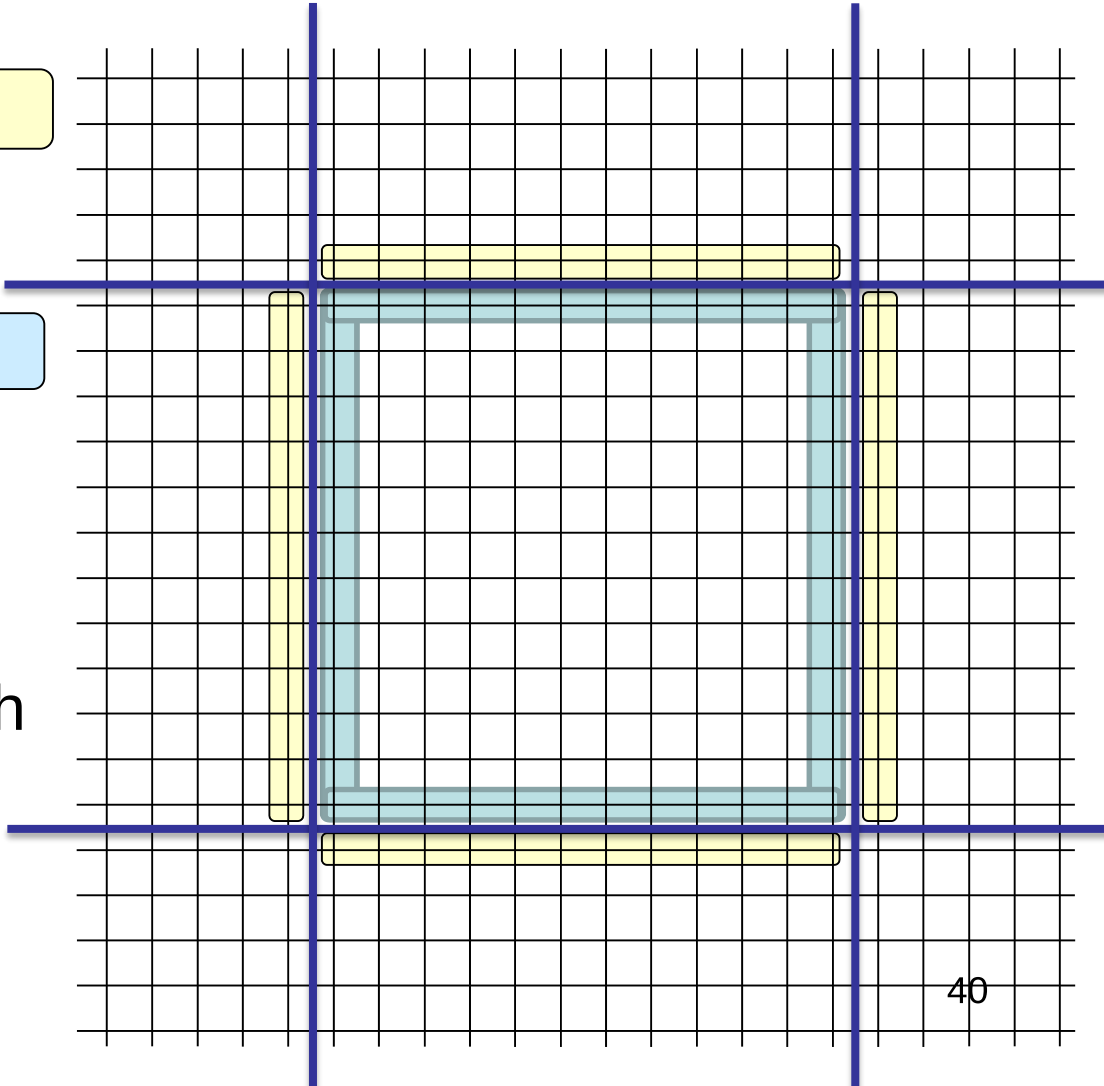


- Block distribution of data enlarges the communication data size
  - In case of 1D n
  - In case of 2D $n/\sqrt{p}$

Communication of shadow region (boundary region)

- To update the boundary 🟦, data of 🟨 is required
- To update the boundary 🟨, data of 🟦 is required

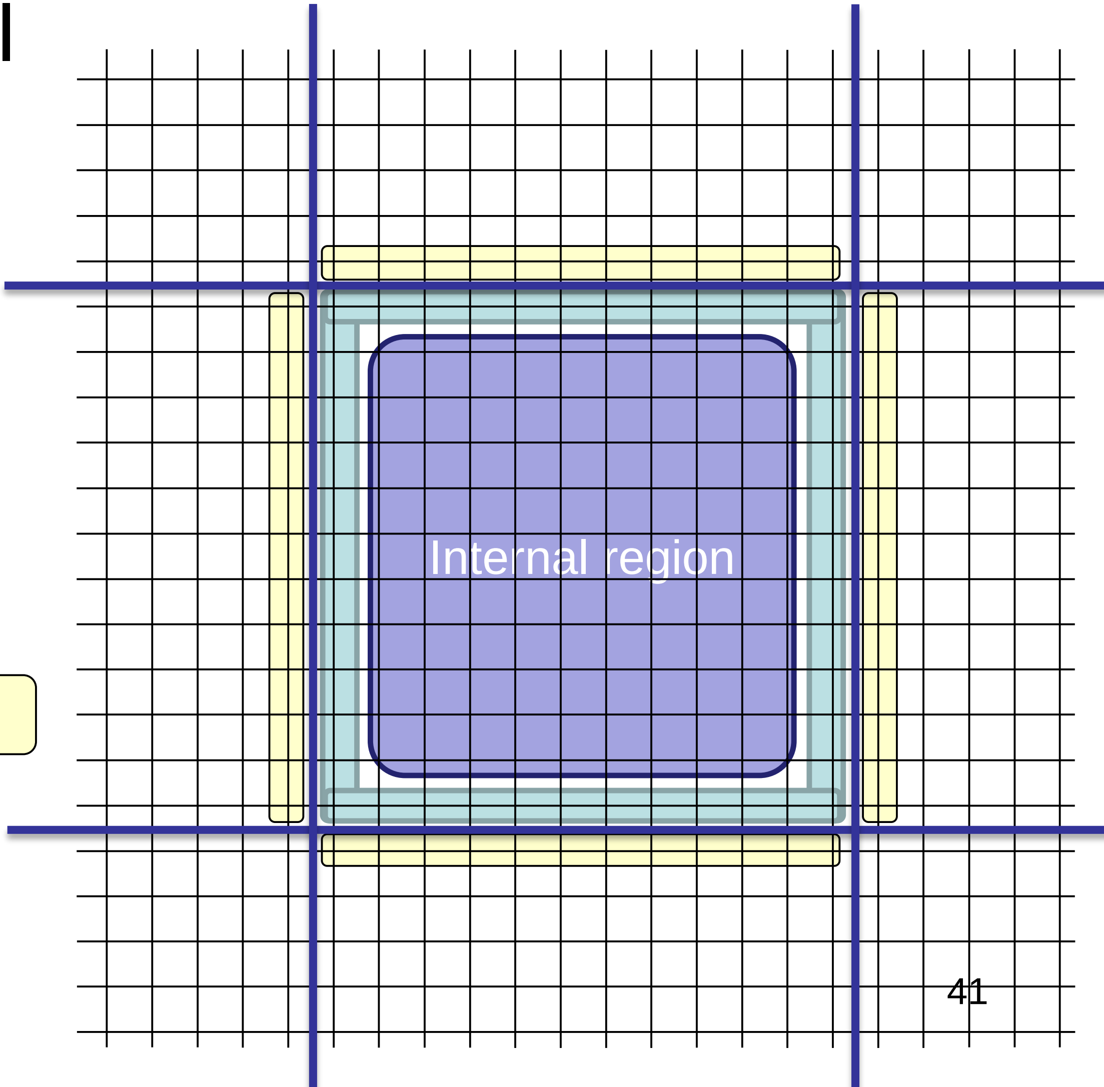
1. Exchange 🟨 and 🟦
2. Update all data in each process



Overlapping computation and communication

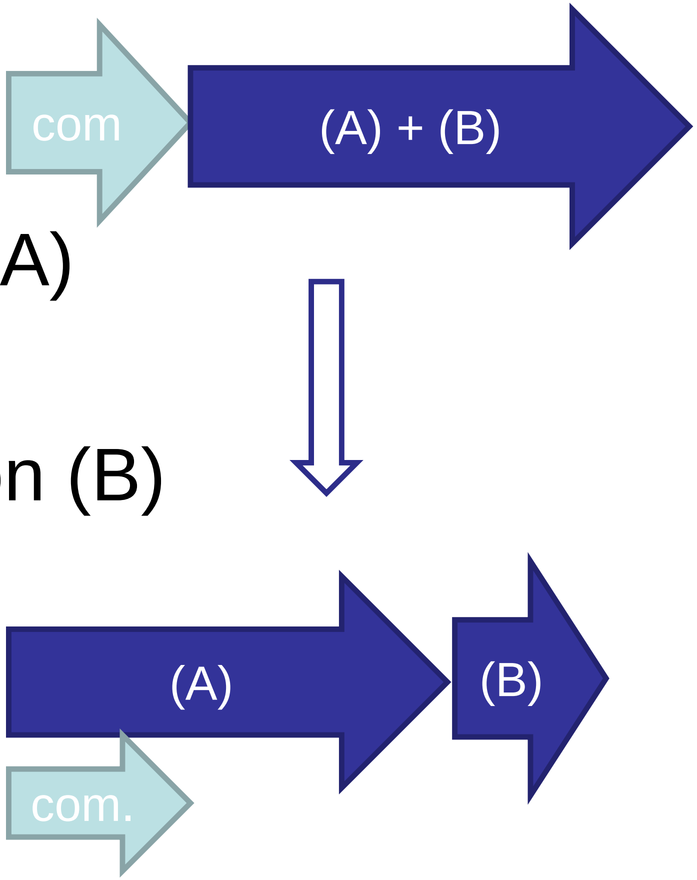
- To update internal region, data of 🟨 is not required

1. Send data of 🟦
2. Update internal region
3. Receive data of 🟨
4. Update boundary region 🟦



Overlapping computation and communication (2)

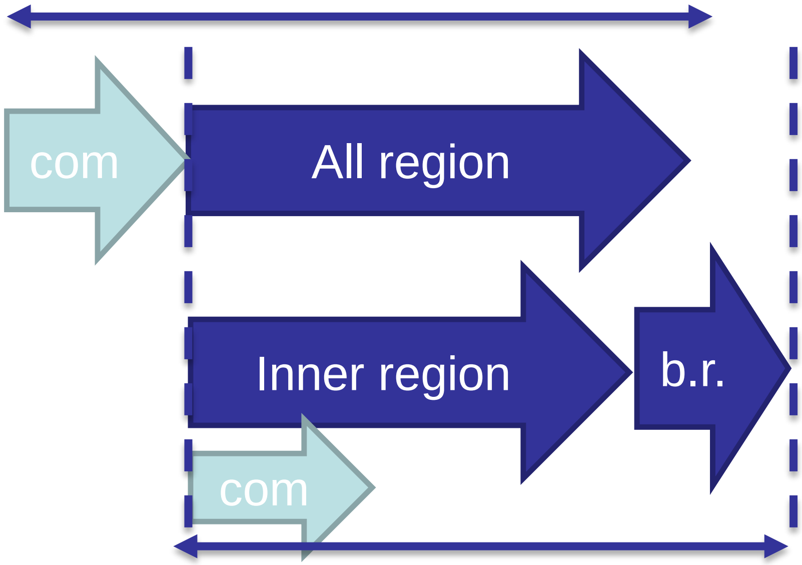
- MPI_Isend(🟦 , …, &req[0])
- MPI_Irecv(🟨 , …, &req[1])
- Calculation in internal region (A)
- MPI_Waitall(2, req, status)
- Calculation on boundary region (B)



Hide communication latency by overlapping computation of internal region and communication

Note for overlapping computation and communication

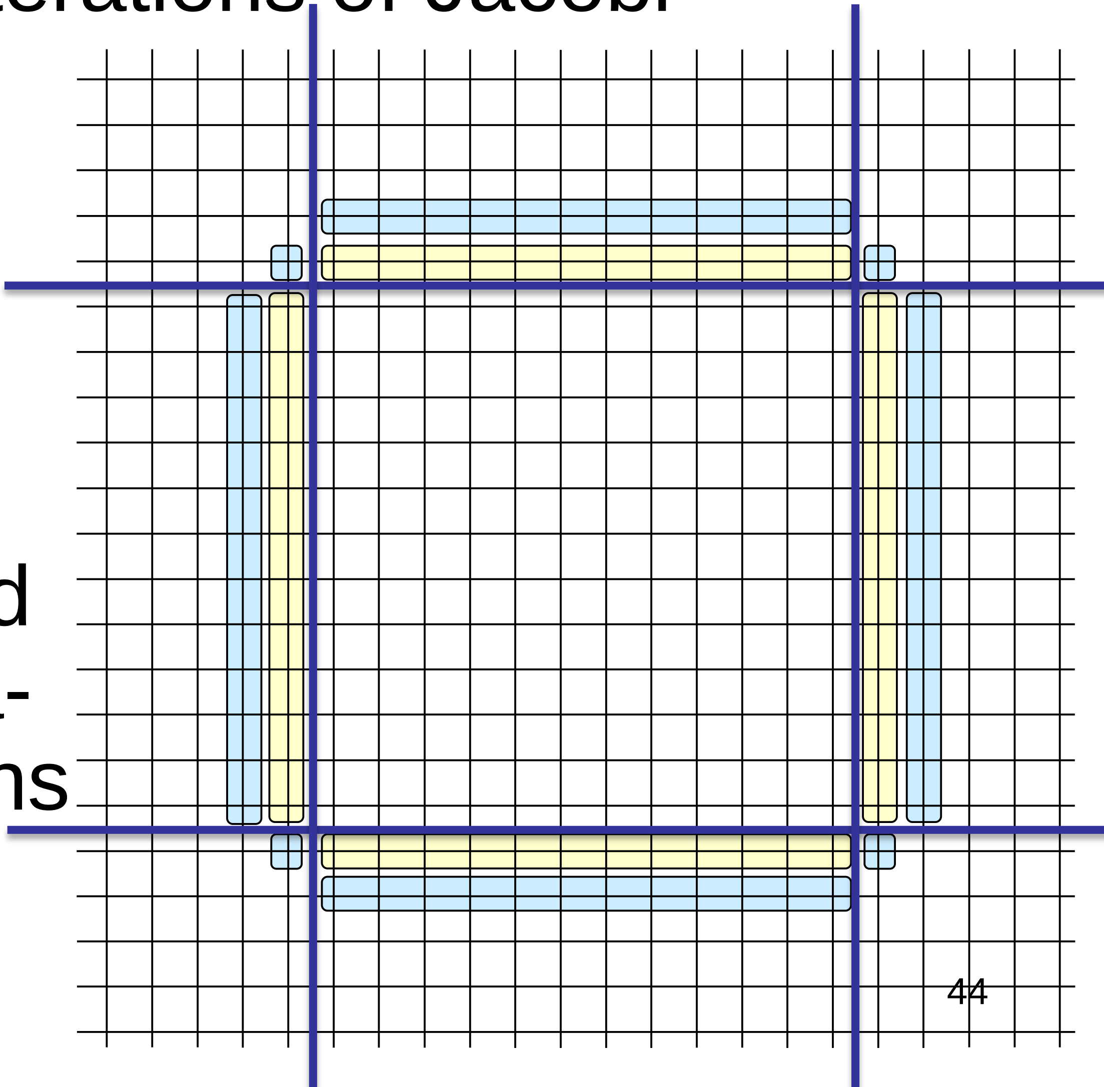
- This may cause the performance degradation
  - Computation of boundary region makes cache miss rate higher
  - Com + all should be less than inner + bound.



Longer computation

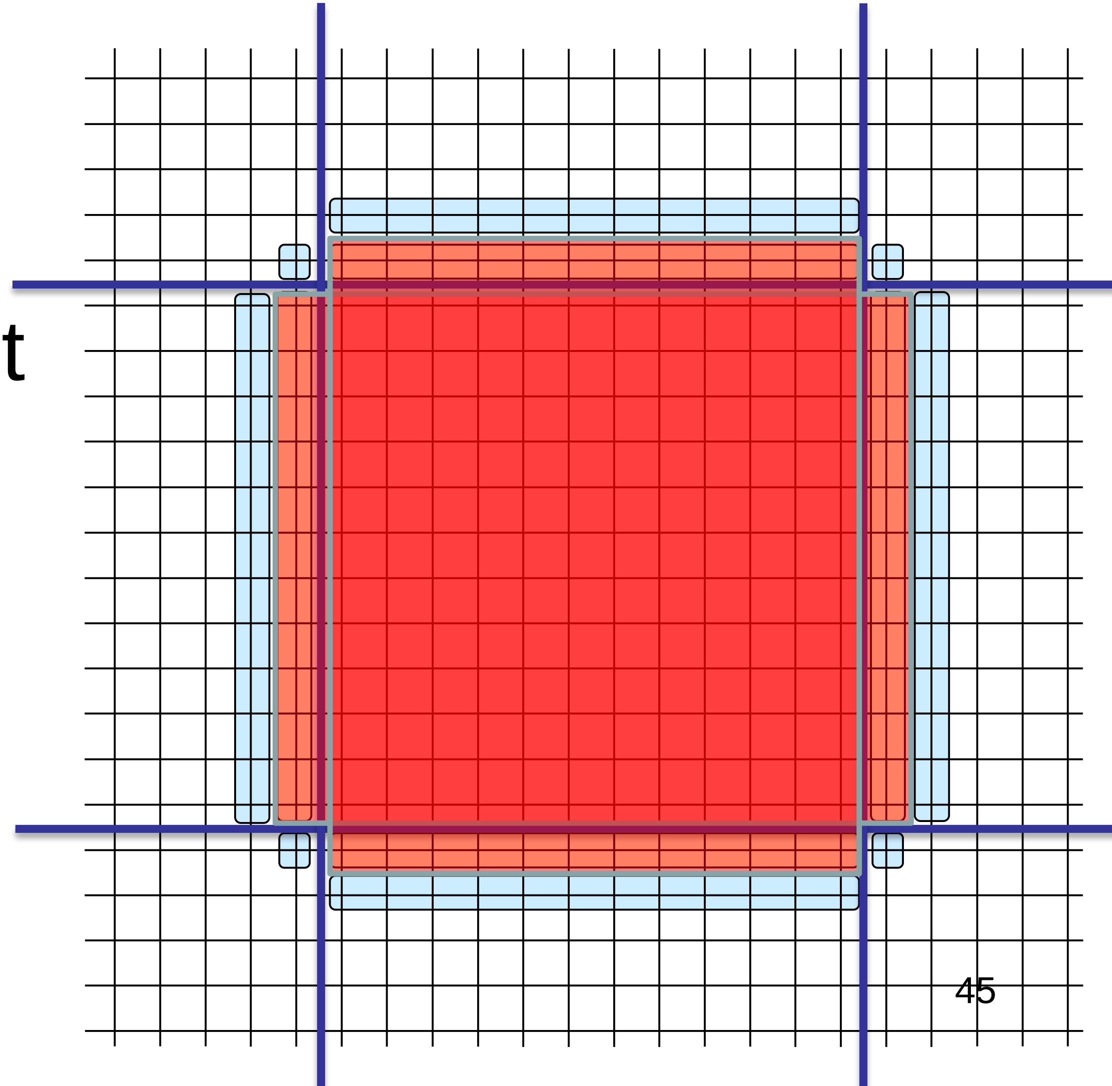
Communication aggregation of multiple iterations (temporal blocking) (1)

- Aggregation of 2 iterations of Jacobi method
- The first iteration requires 🟨
- Next iteration requires 🟦
- Transferring 🟨 and 🟦 enables calculation of two iterations
  - In 1D $2n$
  - In 2D $2n/\sqrt{p}$



Communication aggregation of multiple iterations (2)

- Transfer 🟨 and 🟦
- [First iteration] Compute red part including edge part
- [Second iteration] Compute without communication



Summary

- Basic communication performance
  - Point-to-point communication
  - Collective communication
- profiling
- Communication optimization
  - Communication reduction
  - Communication latency hiding
  - Communication blocking
  - Load balancing