FUJITSU

shaping tomorrow with you

Towards Strong Relationship between Open MPI Community and Fujitsu

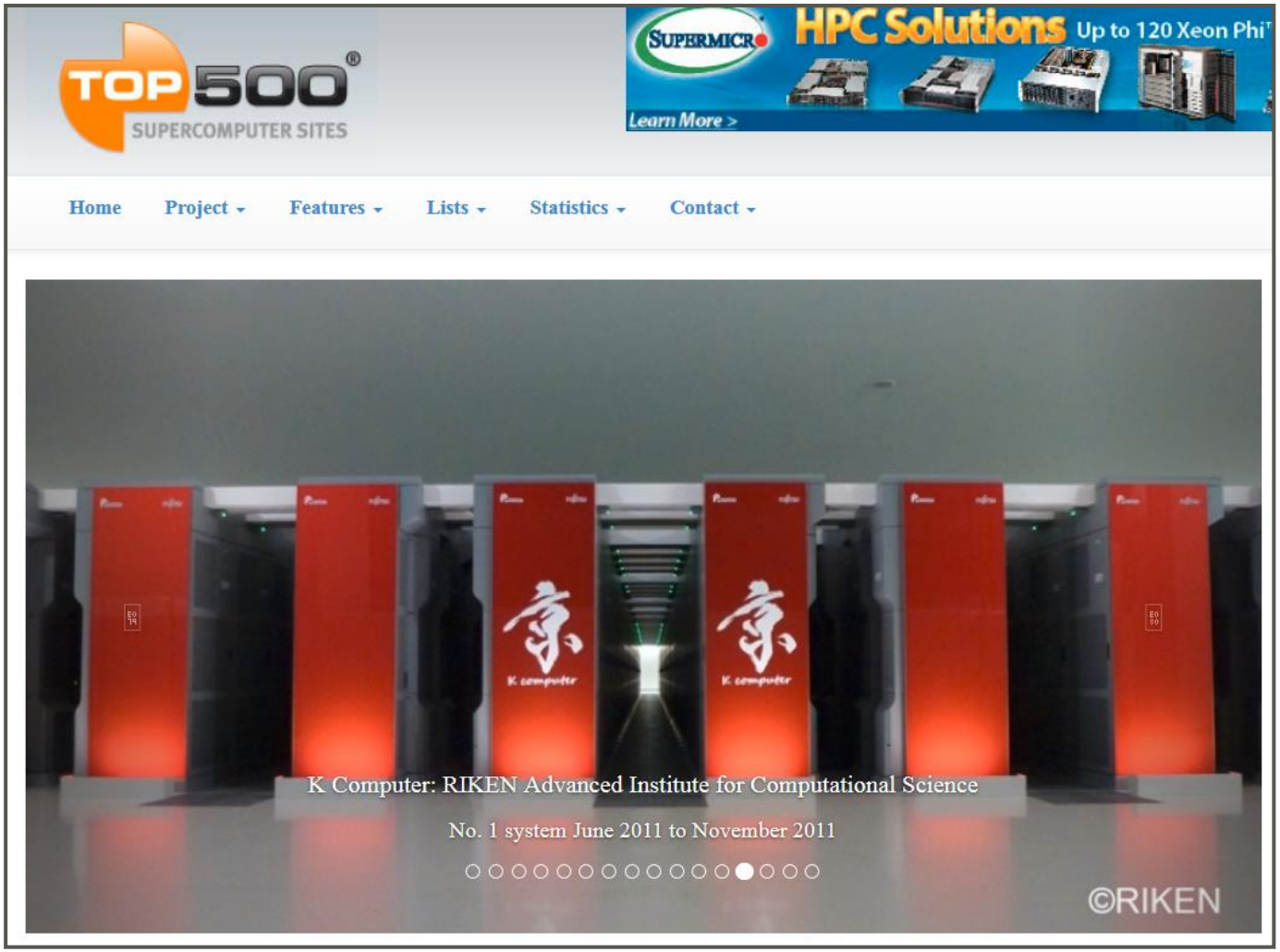
Jan.29, 2015

Computing Language Development Division
Next Generation Technical Computing Unit
Fujitsu Limited

We were #1! > Thanks to Open MPI Team

FUJITSU

- We got the 1st position on TOP500 from June 2011 to November 2011.
  - 10.51 PF (Rpeak 11.28 PF)



http://top500.org/

Outline

- Current Development Status of Fujitsu MPI
- Towards Strong Relationship between Open MPI Community and Fujitsu
  - Third Party Contribution Agreement

FUJITSU

CURRENT DEVELOPMENT STATUS OF FUJITSU MPI

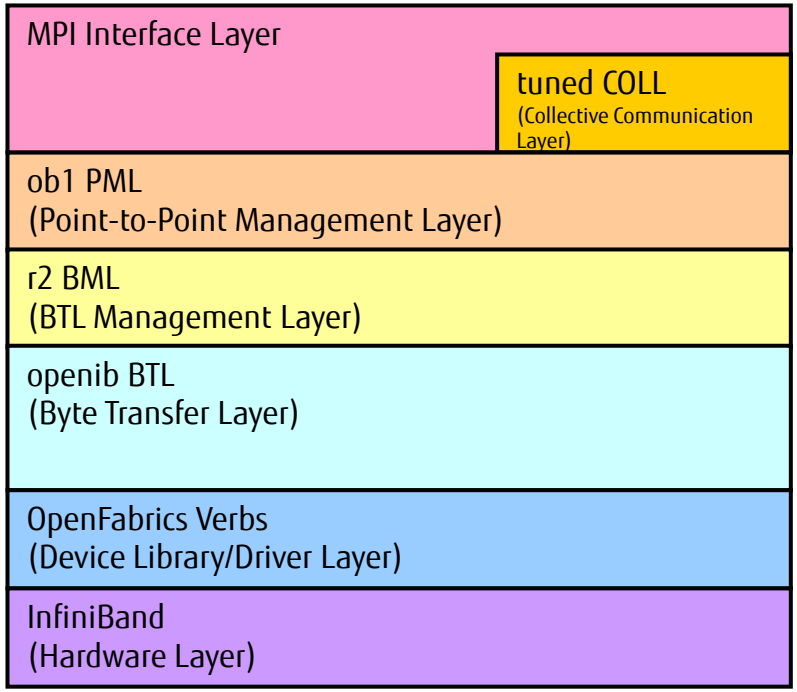
Fujitsu MPI Development

FUJITSU

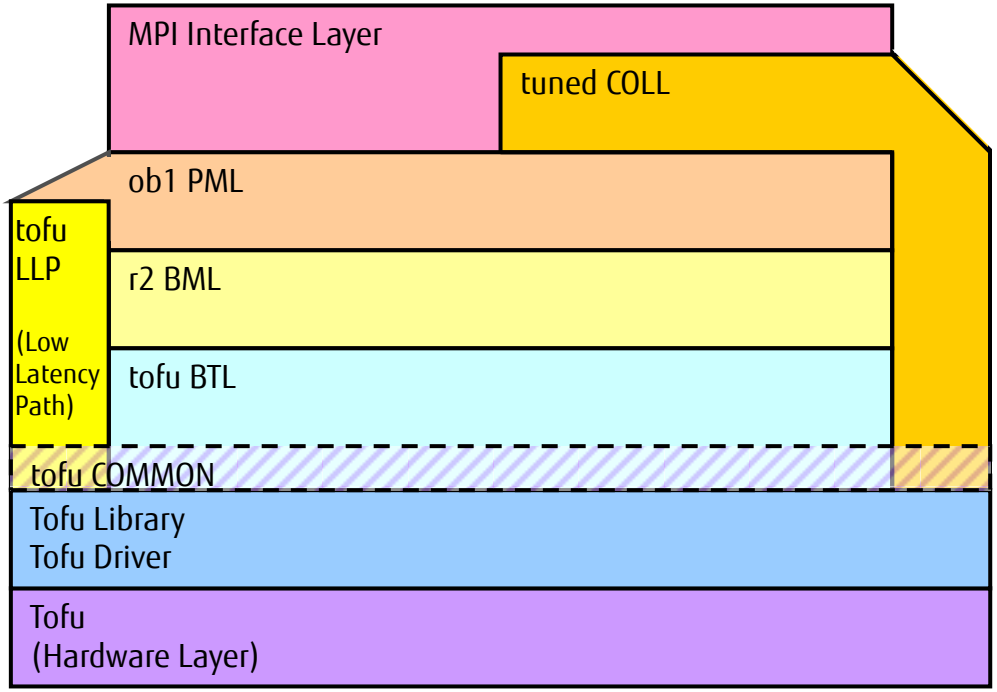
- Fujitsu MPI based on Open MPI
  - The layer structure of Open MPI is excellent.
    - It was easy to support with Tofu interconnect.
  - We developed Fujitsu MPI based on Open MPI.
    - Based on Open MPI 1.4.3
    - Performance improvement
      - a point-to-point communication
      - some collective communication algorithms
      - Tofu Barrier Interface (Barrier, Allreduce, Reduce, Bcast)
    - Support for Fujitsu's process management daemon (use no orted)
- Supported platform
  - K computer, PRIMEHPC FX10 (Tofu)
  - PC Cluster (InfiniBand)

The layer structure

- Fujitsu added new layers
  - tofu LLP (Low Latency Path)
    - Bypass r2 BML and tofu BTL
  - tofu COMMON layer
  - Some collective communication algorithms using RDMA over Tofu

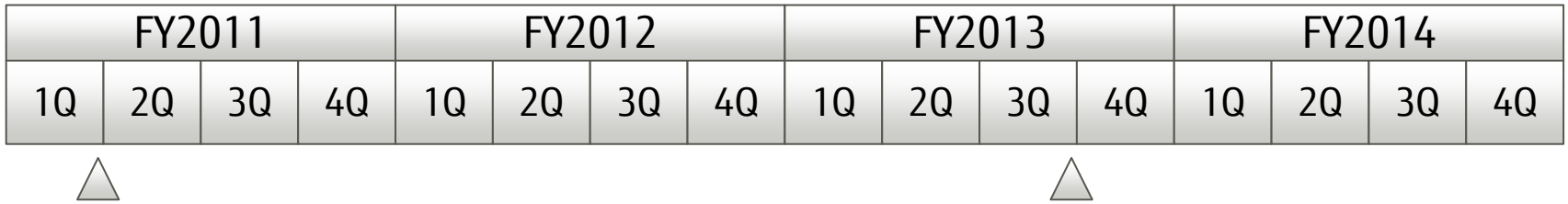


Open MPI



Fujitsu MPI based on Open MPI

MPI Standard conformance



K computer
based on Open MPI 1.4.3

Full MPI-2.1 standard conformance

Technical Computing Suite V1.0L30
based on Open MPI 1.6.1

Full MPI-2.2 standards conformance

Future

FUJITSU

- Fujitsu continues developing based on Open MPI in the future.
  - Fujitsu will release MPI at FY2014/3Q based on Open MPI 1.6.3
    - Full compliance with the MPI-2.2 standards
    - Subset MPI-3.0 standards support
      - mprobe
      - non-blocking collective communications
    - Tofu2 conformance (Tofu2 enhances Tofu)
  - Future (based on Open MPI 1.8.x)
    - Full MPI-3.0 standards support (FY2015/2Q)
  - More future
    - Performance improvement
      - non-blocking collective communications
    - MPI-4.0 standards support

FUJITSU

STRONG RELATIONSHIP BETWEEN OPEN MPI COMMUNITY AND FUJITSU

Third Party Contribution Agreement

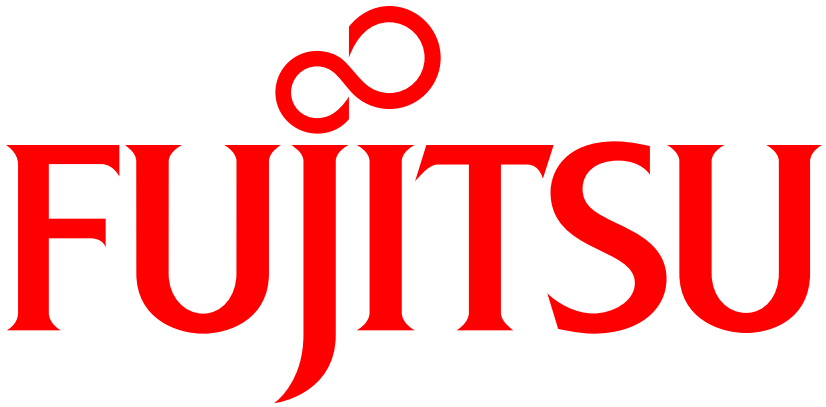
FUJITSU

- Fujitsu will join the Open MPI Development Team soon.
  - Fujitsu will sign Open MPI 3rd Party Contribution Agreement.
    - The FUJITSU MPI development team is consulting the legal section of Fujitsu.
    - Soon... (I hope by the end of Feb.2015)

- Fujitsu would like to cooperate in Open MPI development Team.
  - Develop a new item. (For example, a part of MPI-4.0)
  - Merge the source of the bug fixes and the improvements.

The Open MPI Development Team

FUJITSU



shaping tomorrow with you

FUJITSU

shaping tomorrow with you

MPI Communication Library for Exa-scale Systems

Shinji Sumimoto
Fujitsu Ltd.

Open MPI Developer Meeting, Jan. 2015

Outline of This Talk

FUJITSU

- Our Targets for the Next MPI Development
- Memory Usage Reduction of Open MPI
  - MPI_Init
  - Unexpected Message: Allocator Issue
- Future Development of Fujitsu MPI using Open MPI
  - RDMA Based Transfer Layer for Exa-scale MPI
  - Dynamic Selection Scheme for Collective Communication Algorithm

Our Targets for the Next MPI Development

FUJITSU

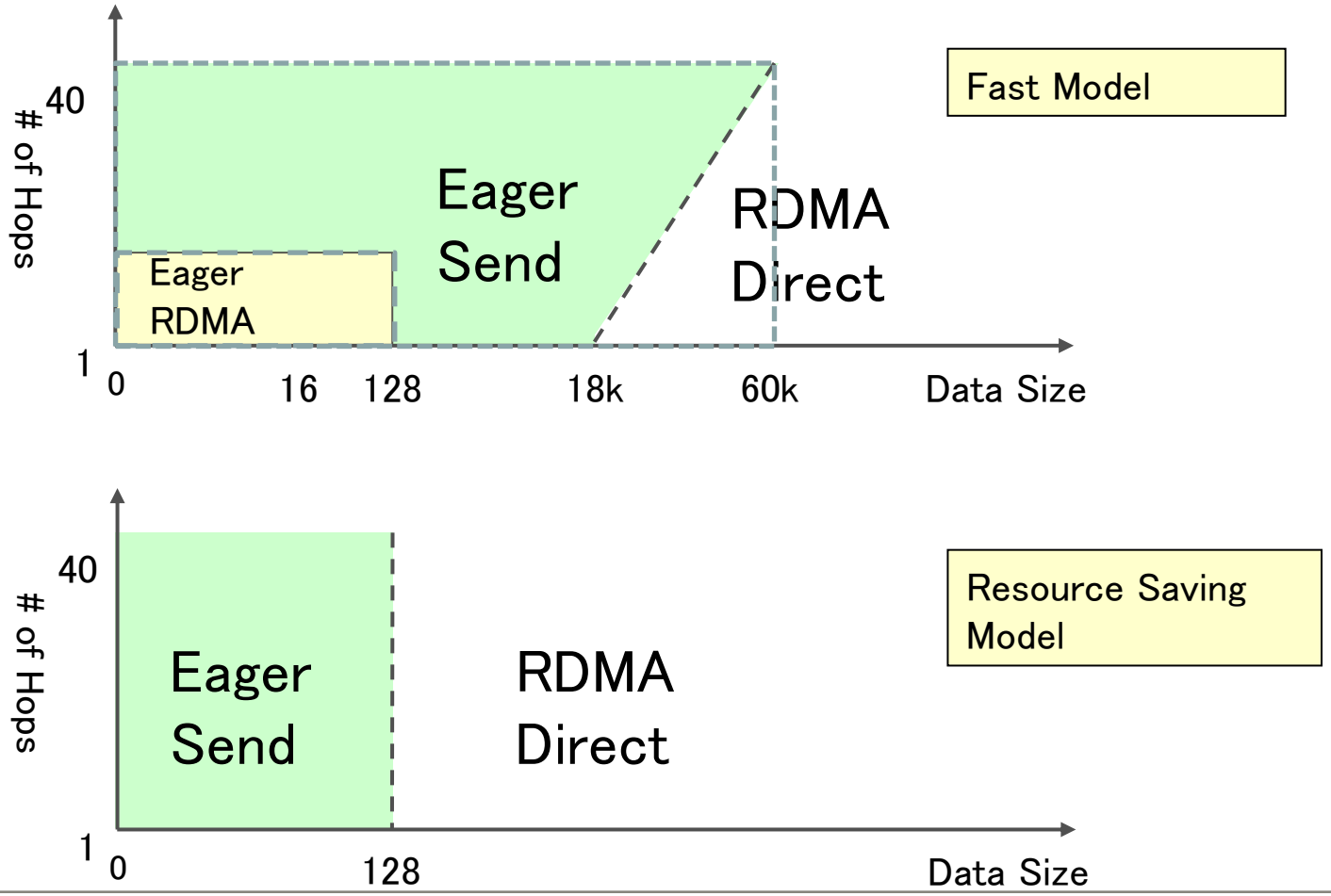
- True Use on Several Million Processes
  - Higher Performance than Current Fujitsu MPI
  - Less Than 1GB Memory Usage Reduction Per Process
- Naturally Integrated MPI Stacks on Open MPI
  - RDMA Based
    - Low Latency Communication Layer
    - Collective Communication by using Multiple RDMAs and Hardware Off-load Engines
- How the Open MPI Communication Layer should be? And, how should Fujitsu contribute to Open MPI Community?
  - Bug Fix, MPI 4.0 (ULFM etc…), Several Options
  - Not Decided yet, we will discuss and propose in this year.

FUJITSU

Memory Consumption Saving Design

Protocol Use Policy of K computer MPI for Resource Saving

- Policy： Providing Fast Model and Resource Saving Model
- Fast Model is used for limited number of destinations
  - User can choose the number of Fast Model Connections.

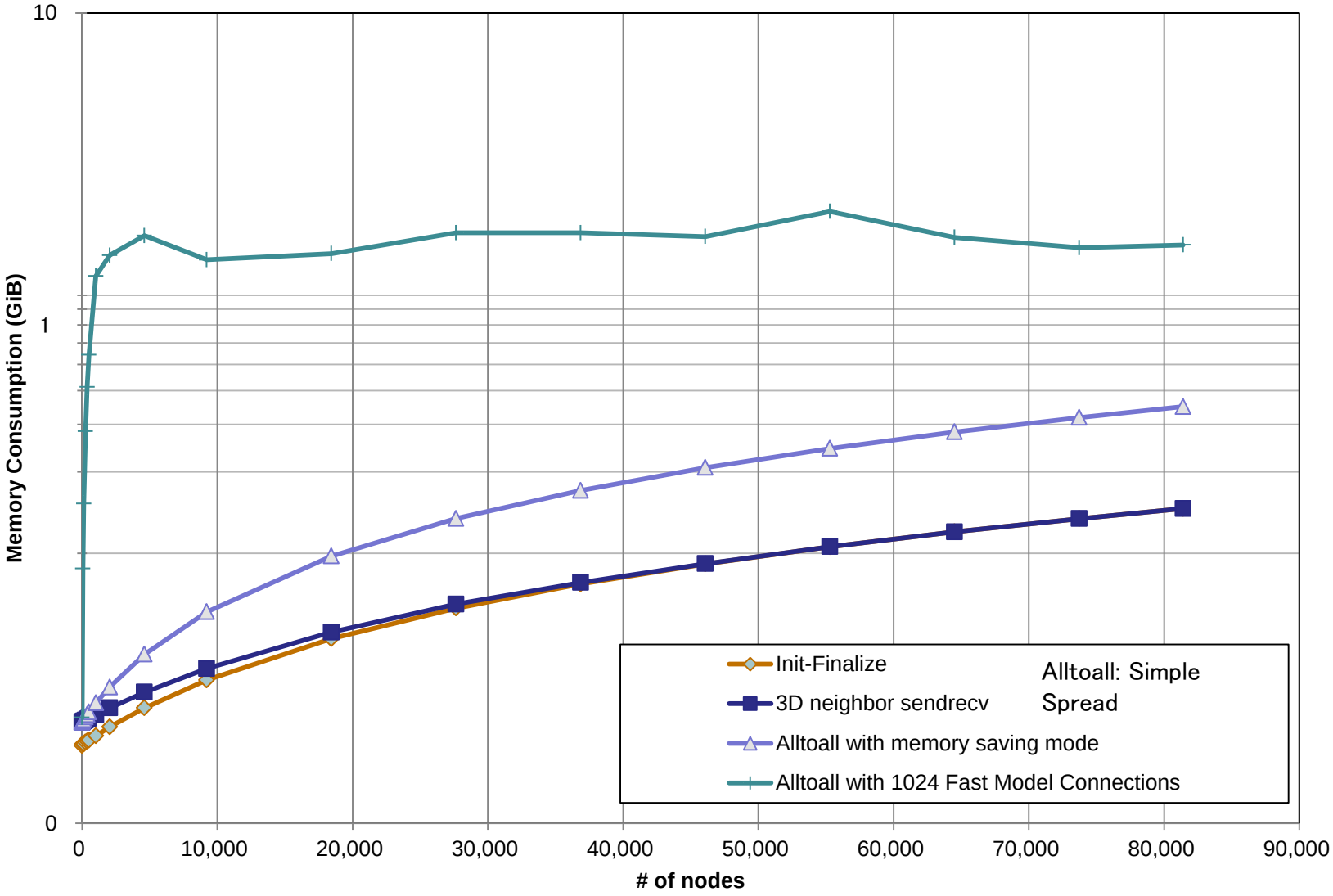


Evaluation of Memory Consumption at Full Scale System

FUJITSU

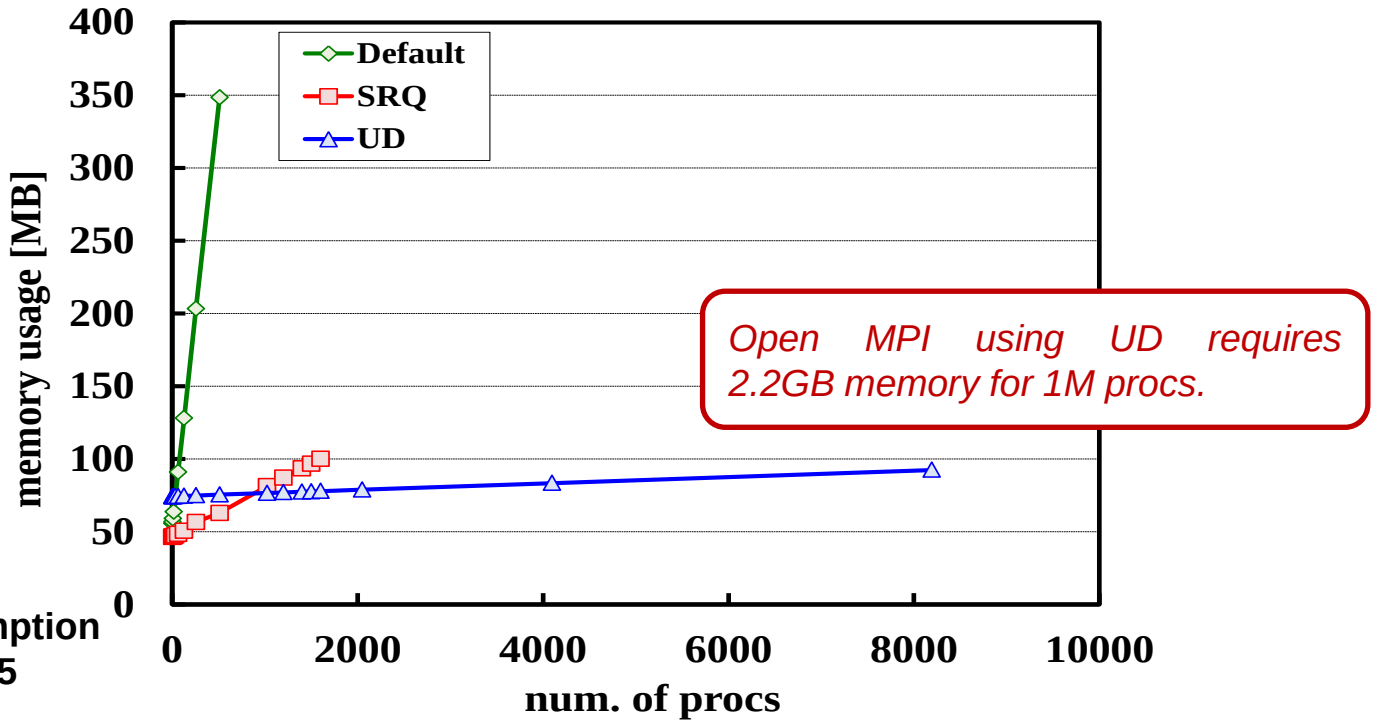
Collaborative work with RIKEN on K computer

- Keeping less than 1.6GB memory usage on full system



Memory Usage Issue

- Post peta-scale system will have 1-10 Millions of Processor Cores
- However, current MPI library requires 2.2GB memory for 1M processes, therefore memory usage of MPI library must be minimized.
- To realize the goal, it is important to know how current MPI library allocates memory.



Memory Consumption of Open MPI 1.4.5 on InfiniBand

Memory Usage of Existing MPI Libraries and Memory Saving Techniques

FUJITSU

- Memory Usage of Existing MPI Libraries: 2 dimensions

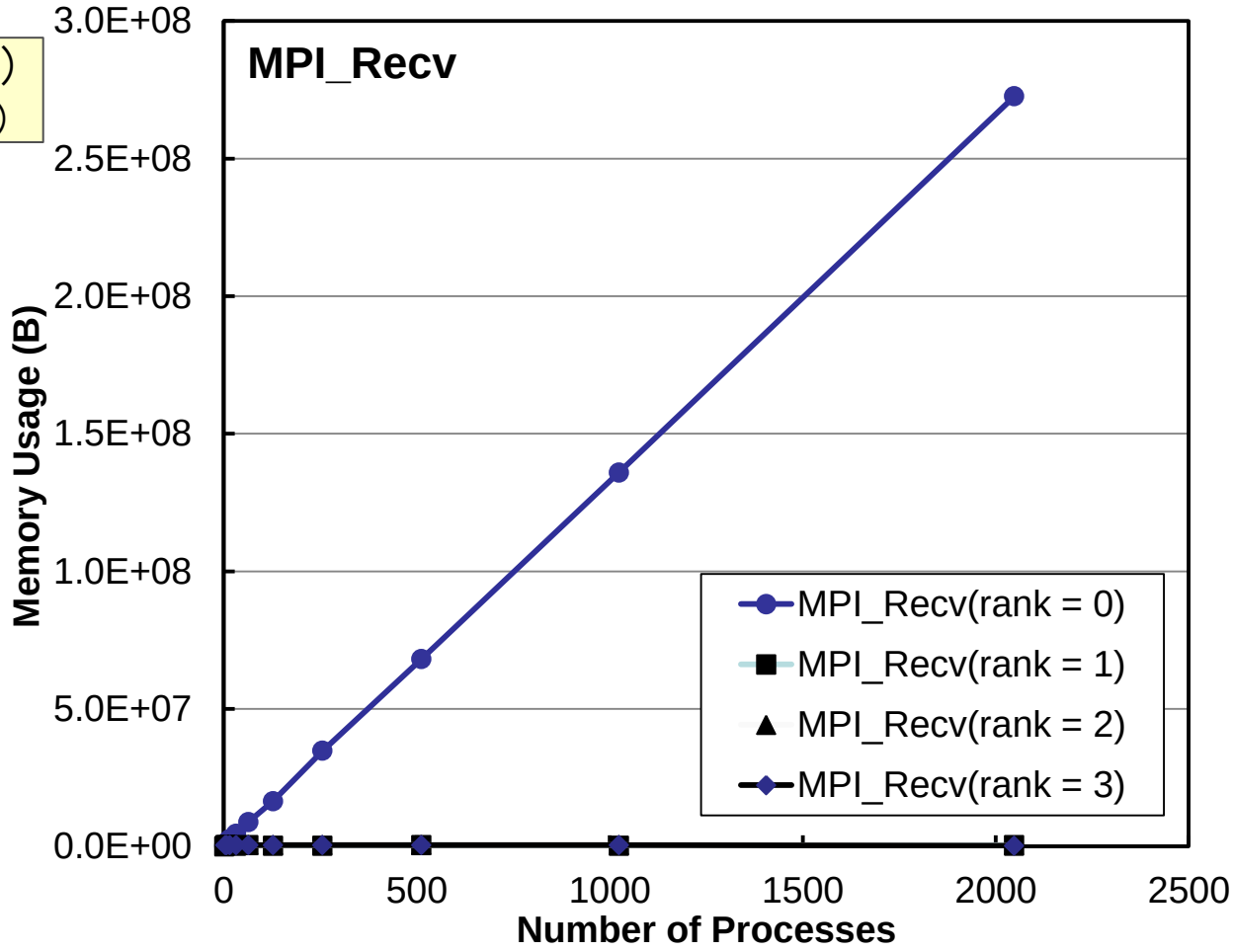
	Device dependent	Device in-dependent
Communication Buffer	Buffer for device etc.	Buffer for collective communication, buffer for Unexpected Message etc.
House Keeping Buffer	Device control structure, command queue, completion queue etc.	Communicator, Tag match table etc.

- Memory Saving Techniques of current MPI Libraries
  - MPICH, Open MPI: Reduction of Device dependent memory of IB
    - RCQP is allocated when a communication starts
    - Shared Receive Queue(SRQ), Unreliable Datagram
  - MPI for K computer:
    - Send/Recv buffer is allocated when a communication starts, Rendezvous + RDMA
    - Selection of High Performance Communication and Saving Memory Communication Modes
- Memory Saving of House Keeping Buffer is out of scope.

Memory Usage of MPI_Recv: IMB(MPI_Exchage)

FUJITSU

o(1K): o(100MB)
o(1M): o(100GB)

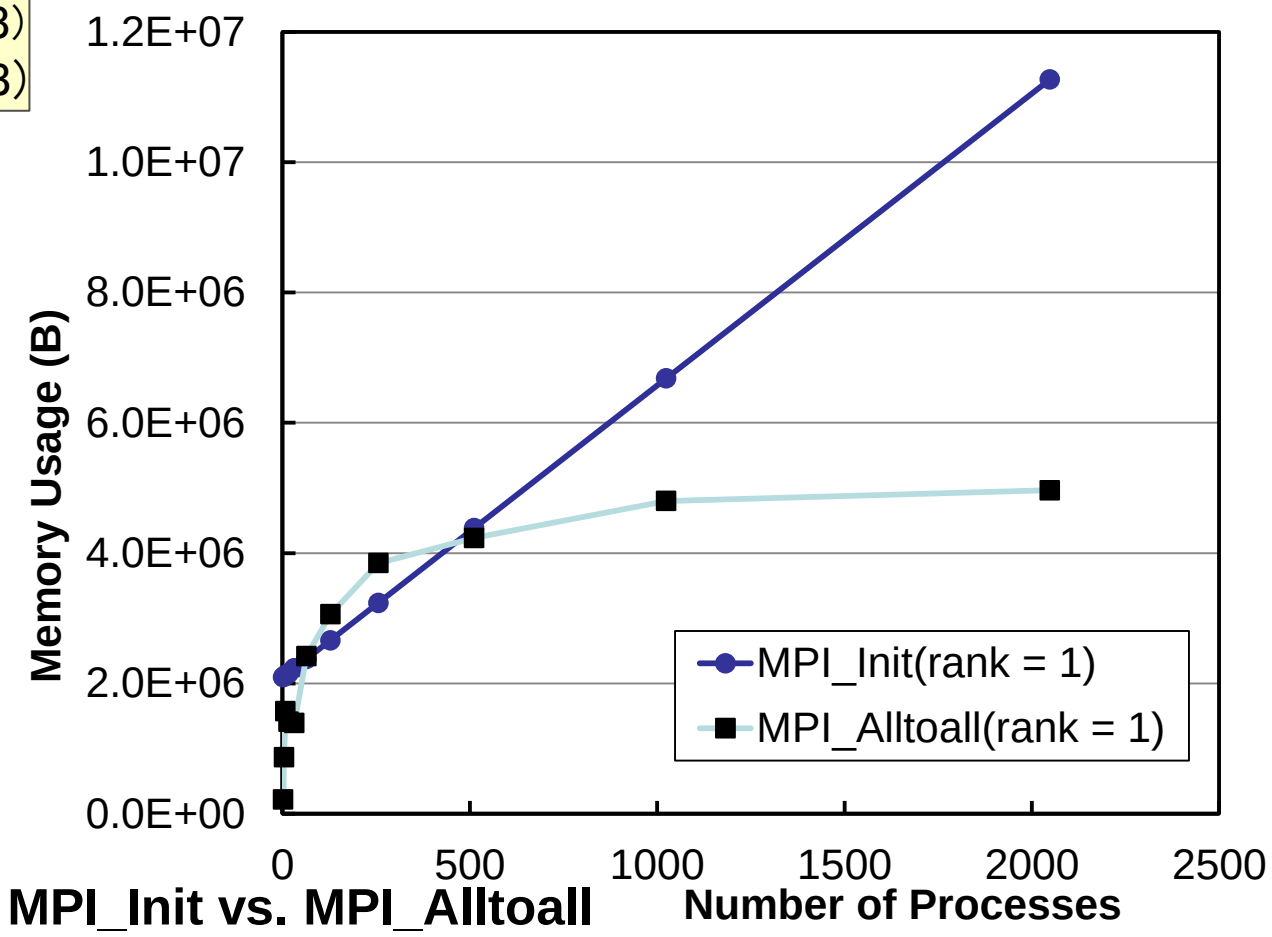


- Only Memory Usage of Rank0 increases
- The Reason is Unexpected Message which the other processes sent to rank0

Memory Usage MPI_Init vs. MPI_Alltoall: Rank1

FUJITSU

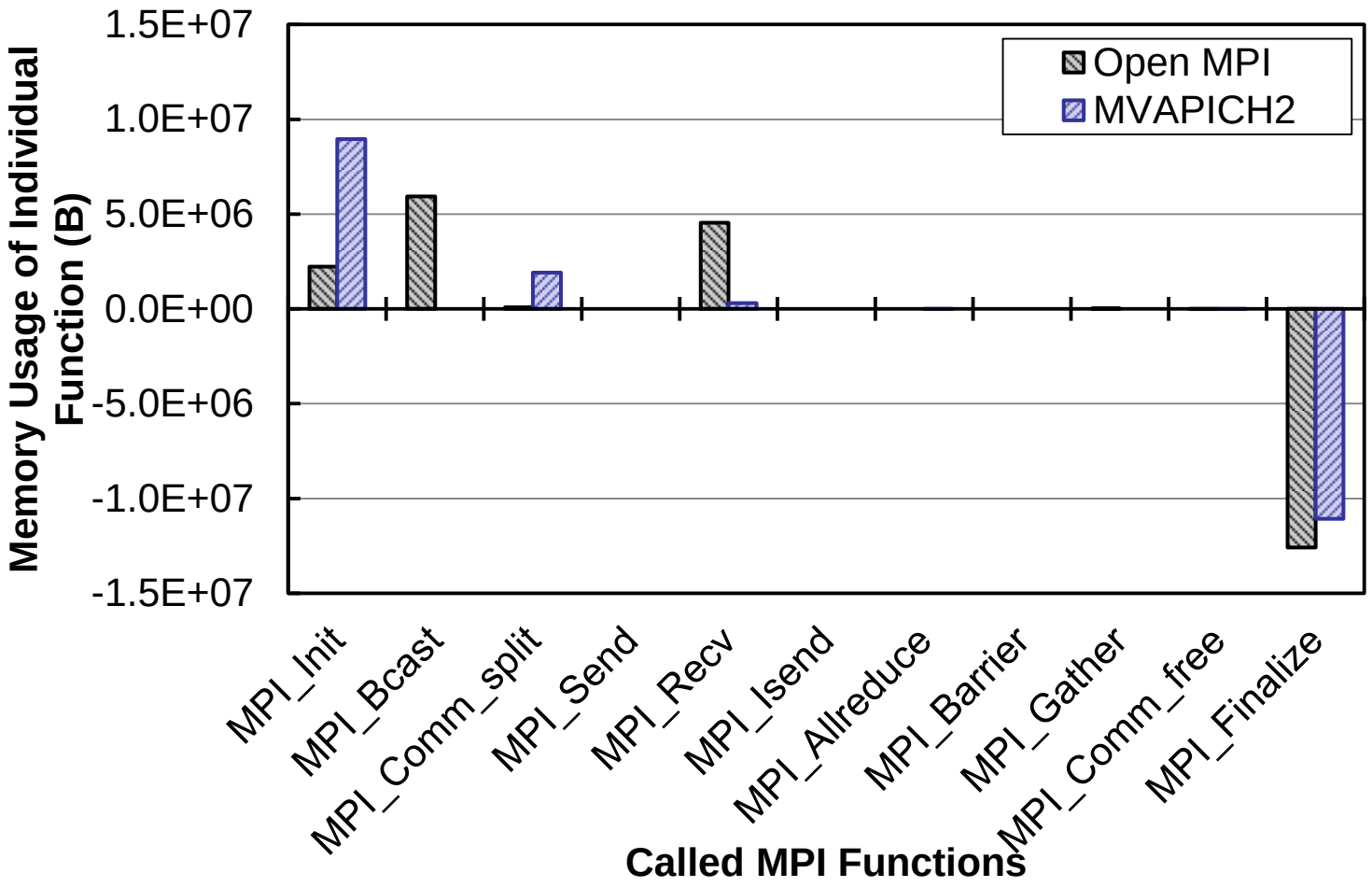
o(1K): o(1−10MB)
o(1M): o(1−10GB)



- Memory usage of MPI_Init is larger than that of MPI_Alltoall
  - Memory usage of MPI_Init should be analyzed in more details

Memory Usage Open MPI vs. MVAPICH2

FUJITSU

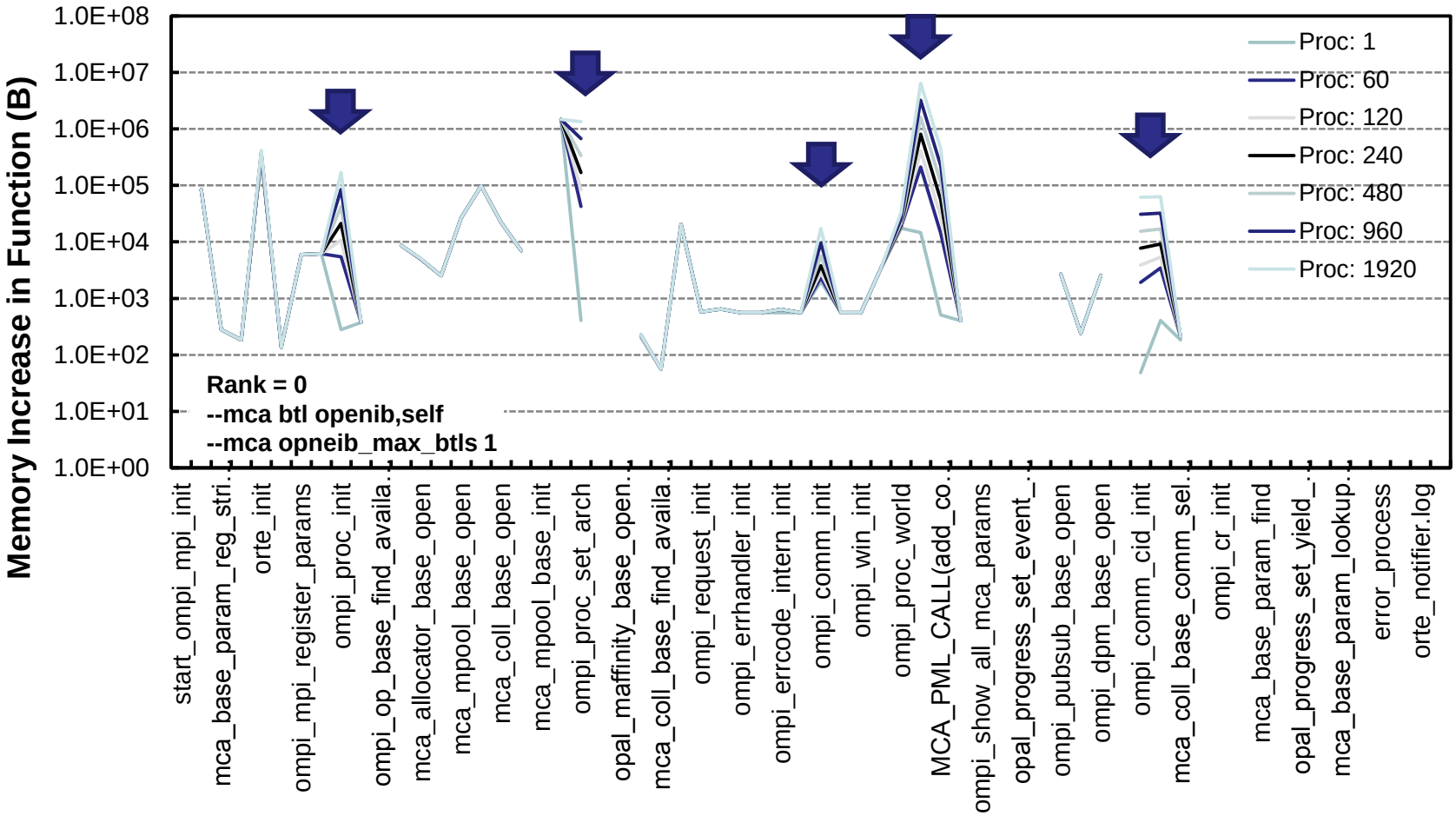


- Both of Open MPI and MVAPICH2 free at MPI_Finalize
- MVAPICH2 allocates memory at MPI_Init, and Open MPI allocates memory at MPI communication

Memory Usage of Functions in MPI_Init Using Snapshot Function

FUJITSU

- Measuring by Increasing number or process（1 ～ 1920）

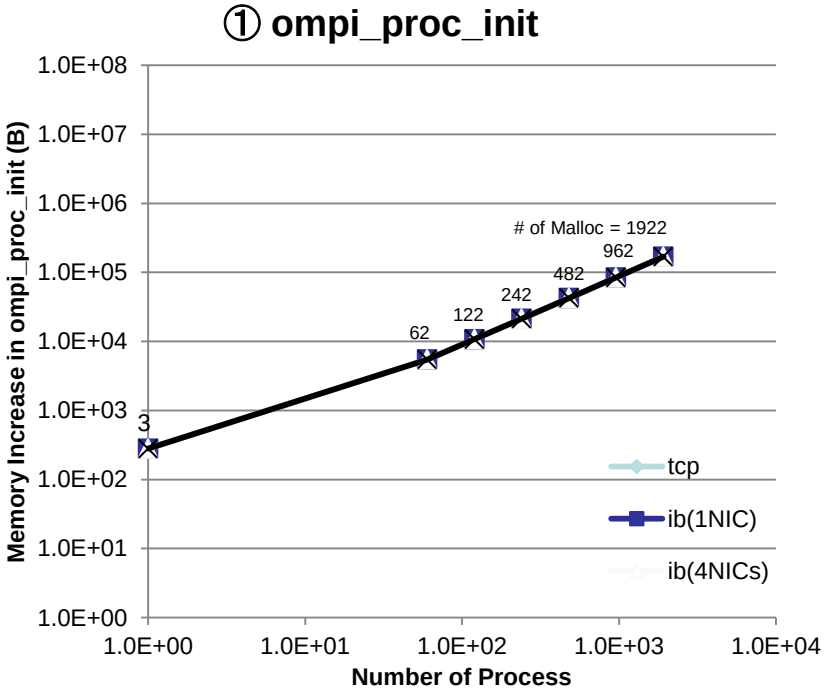


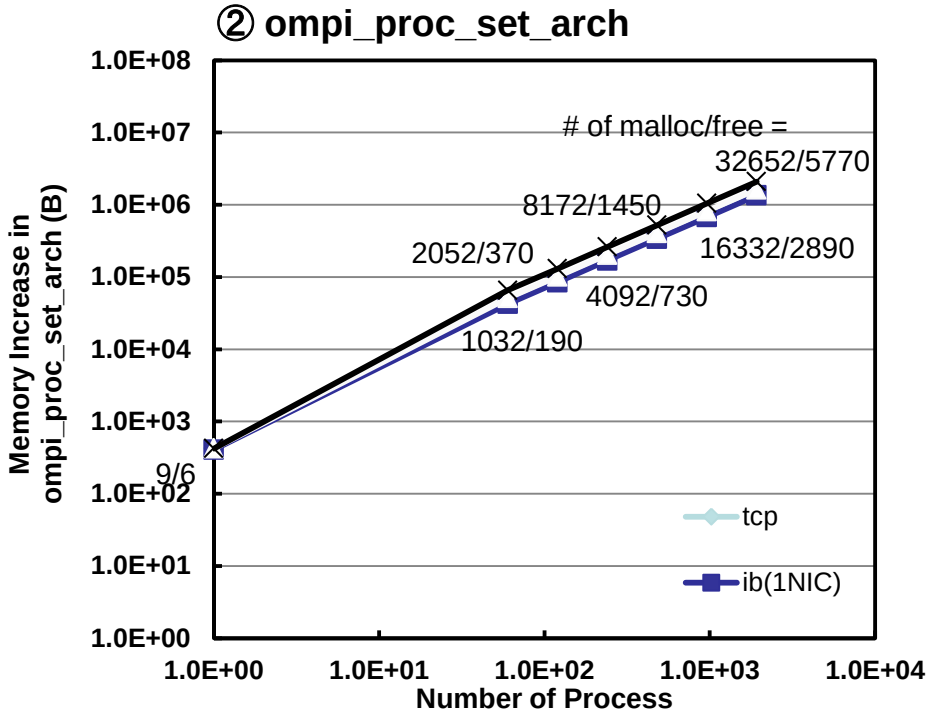
Several functions use array tables and linked list of structures, and the data includes other process information redundantly.

FUJITSU

WHY CURRENT COMMUNICATION LIBRARIES NEED SO MUCH MEMORY

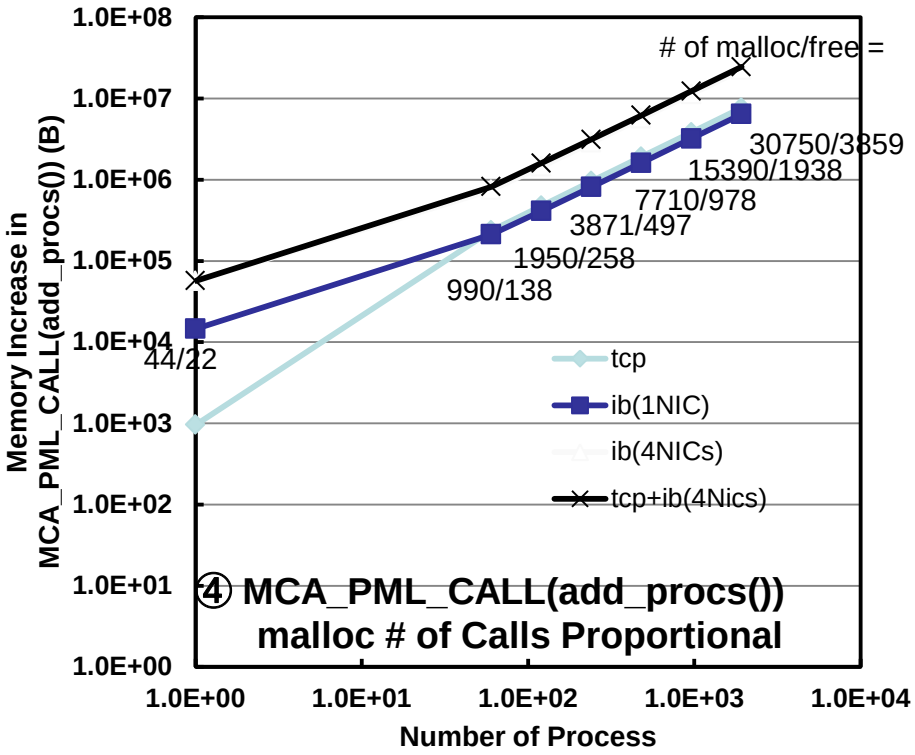
Memory Usage Analysis : Proportional to Number of malloc Calls





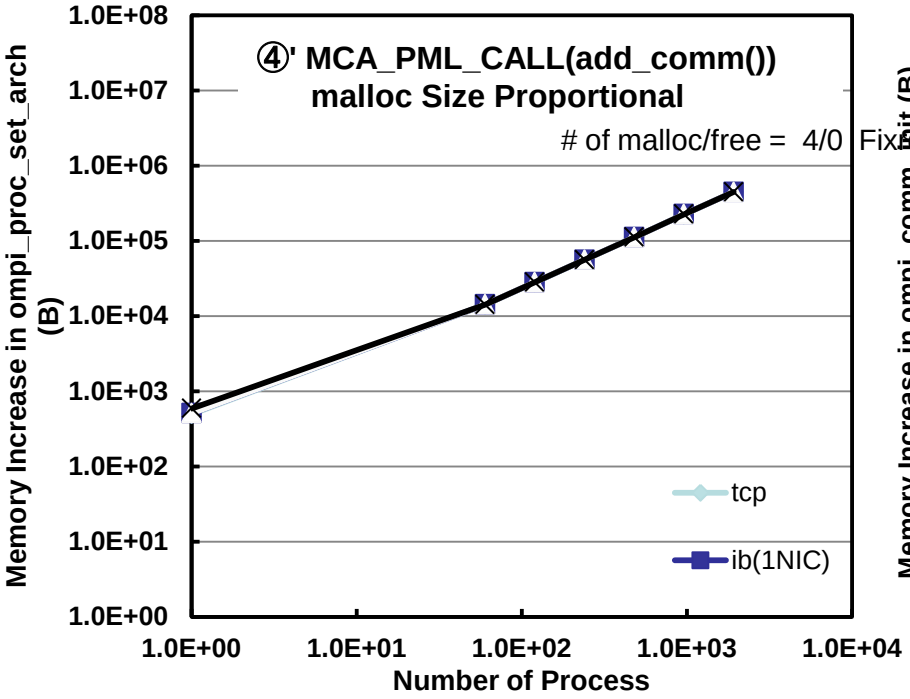
- Functions which proportional to number of malloc calls by increasing number of process
  - ompi_proc_init： Device Independent, House Keeping
  - ompi_proc_set_arch: Device Dependent, House Keeping

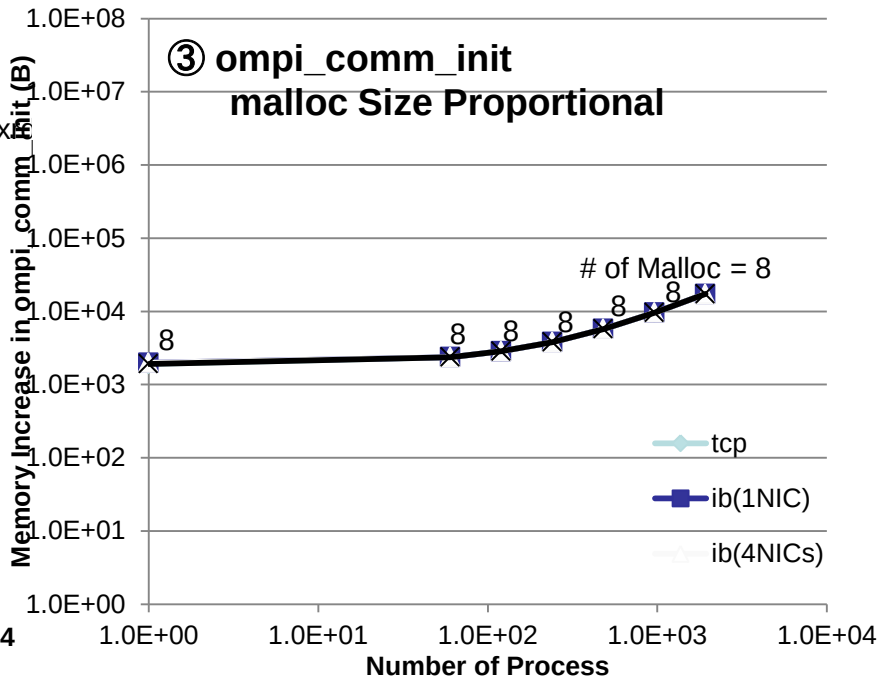
Memory Usage Analysis : Proportional to Number/Size of malloc Calls(2)



- Functions which proportional to number of malloc calls/size by increasing number of process
  - MCA_PML_CALL(add_procs()) : Device Dependent, House Keeping

Memory Usage Analysis : Proportional to Size of malloc Calls

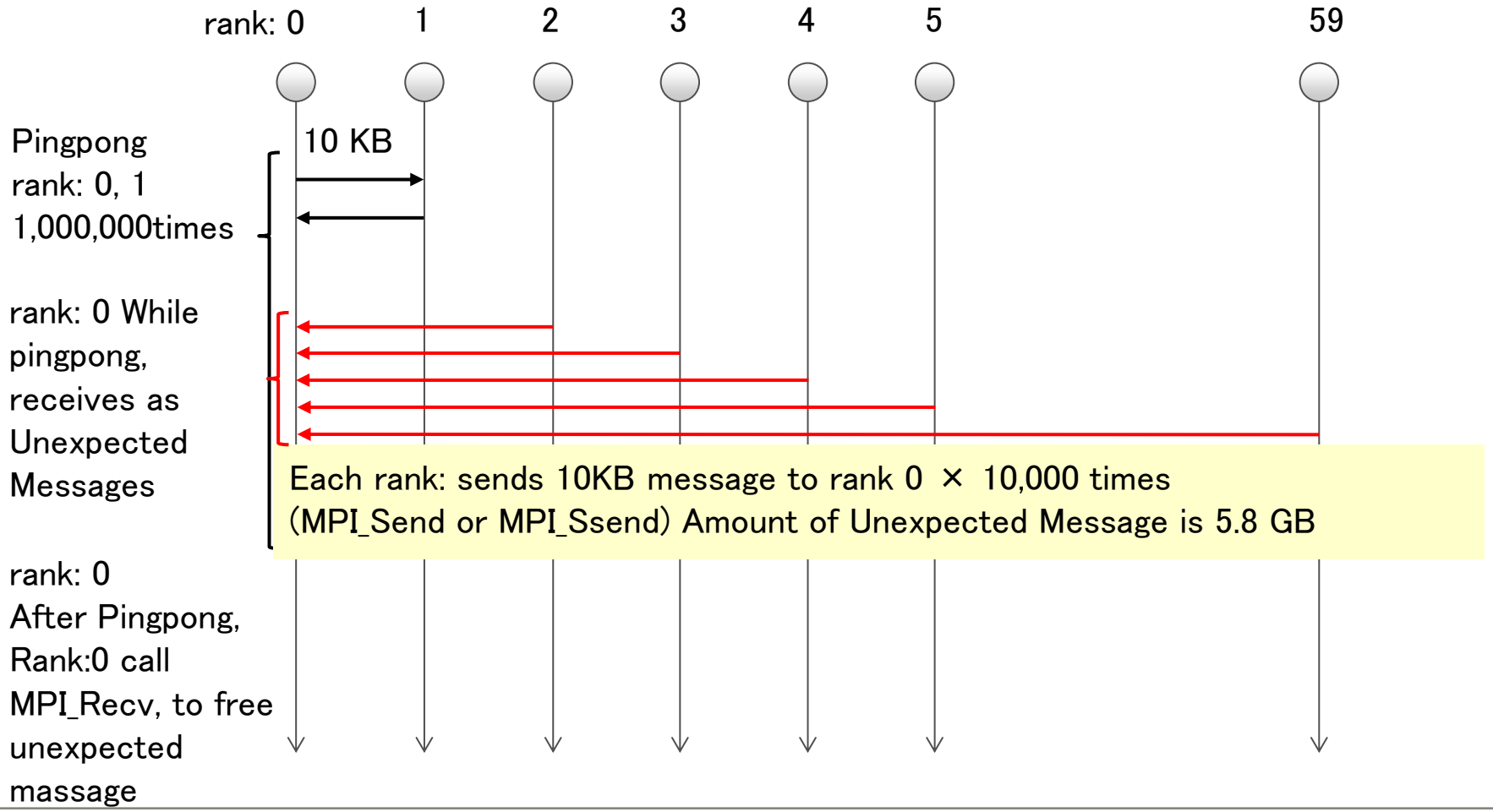




- Functions which proportional to malloc size by increasing number of process
  - MCA_PML_CALL(add_comm()): Device Independent, House Keeping
  - ompi_comm_init: Device Independent, House Keeping

Memory Usage Test of Unexpected Messages

- While rank0-1 do pingpong, rank2-59 send message to rank0
- After rank0-1 finish pingpong, rank0 receives message of rank2-59
- Comparing memory usage of MPI_Send（Eager） and MPI_Ssend（Sync.）



Unexpected Message Test Results

FUJITSU

- MPI_Send
  - Memory Usage: 10,188,099,848 Bytes(9.488GB)
  - Reason for 9.488GB not 5.8GB：Rounds up 10KB message to 16KB area (9.155GB)
- MPI_Ssend（sync. send）
  - Memory Usage：30,822,896 Bytes(0.028GB)
- Memory is freed at MPI_Finalize： This means Open MPI does not free allocated memory untile MPI_Finalize

MPI_Send: Unexpected Message

```
---- Statistics of individual library memory usage -------
Library: /home/akimoto/OpenMPI/lib/libmpi.so.1
mem_size = 933304, mem_min = 0, mem_max = 10198882832
malloc: 630673, realloc: 832, memalign: 2074, free: 628654
  ---- Statistics of individual function memory usage ----
  Function: MPI_Init
  mem_size = 8761800, mem_min = 0, mem_max = 8777336
  malloc: 24456, realloc: 831, memalign: 38, free: 15420
  Function: MPI_Send
  mem_size = 10188099848, mem_min = 0, mem_max = 10188100488
  malloc: 603541, realloc: 0, memalign: 2036, free: 4114
  Function: MPI_Recv
  mem_size = 1952728, mem_min = 0, mem_max = 1952728
  malloc: 113, realloc: 0, memalign: 0, free: 0
  Function: MPI_Finalize
  mem_size = -10197949056, mem_min = -10197949056, mem_max = 472
  malloc: 1131, realloc: 1, memalign: 0, free: 608832
  ------------------------------------------------------
```

MPI_Ssend: No Unexpected Message

```
---- Statistics of individual library memory usage -------
Library: /home/akimoto/OpenMPI/lib/libmpi.so.1
mem_size = 928544, mem_min = 0, mem_max = 76597232
malloc: 41417, realloc: 832, memalign: 2042, free: 39371
  ---- Statistics of individual function memory usage ----
  Function: MPI_Init
  mem_size = 8756528, mem_min = 0, mem_max = 8772064
  malloc: 24448, realloc: 831, memalign: 38, free: 15417
Function: MPI_Send
  mem_size = 30822896, mem_min = 0, mem_max = 30823536
  malloc: 3852, realloc: 0, memalign: 530, free: 1070
  Function: MPI_Recv
  mem_size = 36949280, mem_min = 0, mem_max = 36949824
  malloc: 10576, realloc: 0, memalign: 1474, free: 3022
  Function: MPI_Finalize
  mem_size = -75668144, mem_min = -75668144, mem_max = 472
  malloc: 1109, realloc: 1, memalign: 0, free: 19574
  ------------------------------------------------------
```

Investigating of Open MPI(1.4.6) Source (same as ver. 1.8.4)

FUJITSU

```
ompi/mca/pml/ob1/pml_ob1_recvfrag.h
#define MCA_PML_OB1_RECV_FRAG_RETURN(frag) ¥ do { ¥
            if( frag->segments[0].seg_len > mca_pml_ob1.unexpected_limit ){ ¥
             /* return buffers */ ¥
            mca_pml_ob1.allocator->alc_free( mca_pml_ob1.allocator, ¥
                  frag->buffers[0].addr ); ¥
             } ¥
            frag->num_segments = 0; ¥
             ¥
            /* return recv_frag */ ¥
            OMPI_FREE_LIST_RETURN(&mca_pml_ob1.recv_frags,¥
                  (ompi_free_list_item_t*)frag); ¥
} while(0)
```

- Open MPI frees unexpected message when the message size is larger than mca_pml_ob1.unexpected_limit
- But allocator function of mca_pml_ob1.allocator->alc_free() does not free the memory. This fact needs to change allocator functions.
- In case of bucket allocator, mca_allocator_bucket_cleanup frees the memory. Easy to implement.

Memory Saving Method Direction of MPI (Open MPI)

FUJITSU

- MPI_Init
  - Memory Usage for House Keeping is proportional to number of process.
  - Memory Saving Technique must be applied to both of device dependent/independent ways
- Unexpected Message
  - Unexpected message is not freed until MPI_Finalize
  - Some Limitation must be implemented

FUJITSU

Collective Communication Design

From the paper, “The Design of Ultra Scalable MPI Collective Communication on the K Computer”

Tomoya Adachi Fujitsu, ISC12 Research Paper

Design Policy of Collective Algorithms

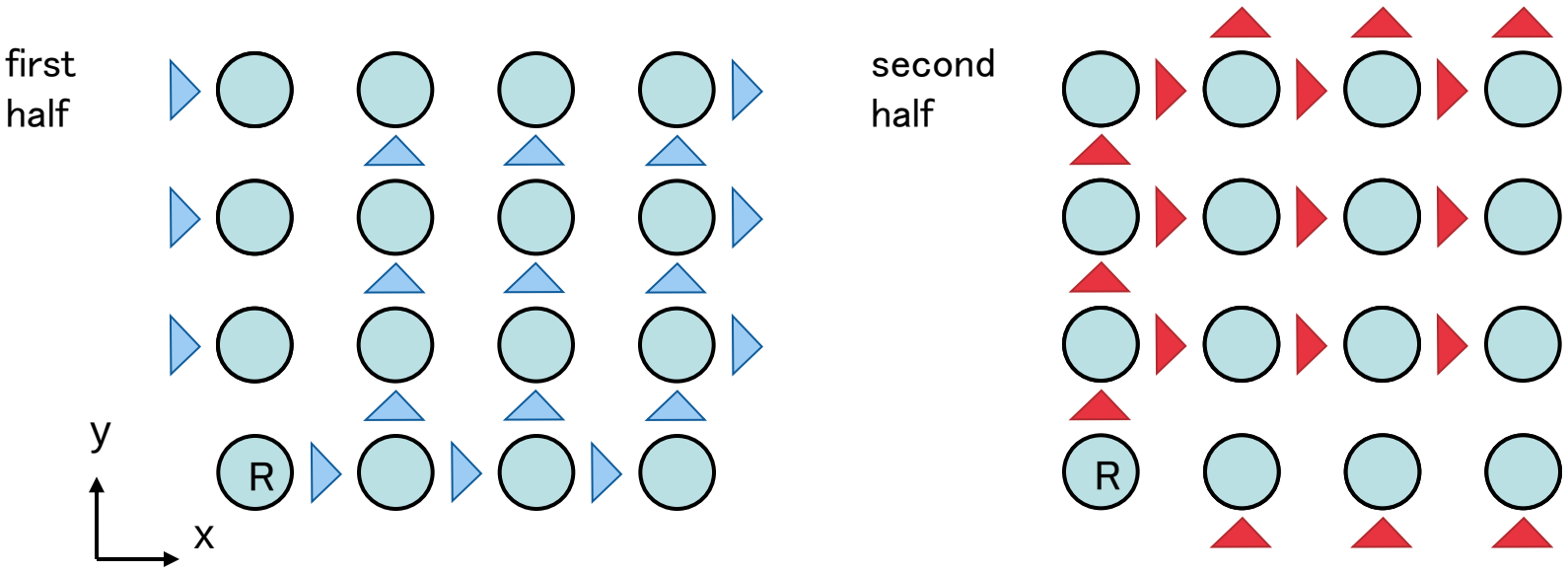
FUJITSU

- Long-message algorithms: high throughput
  - Multi-NIC-awareness and collision-freeness
  - →Pipeline transfer along multiple edge-disjoint paths
  - →Communicating only with neighbor nodes
- Short-message algorithms: low latency
  - Relaying cost (both software & hardware) is un-ignorable
  - →Reducing the number of relaying nodes (*steps*)

    Note: from the user's point of view, which algorithm to use is automatically determined in accordance with the message size and # of processes
- Whole tuned algorithms are implemented using only RDMA
  - To minimize memory footprint of intermediate buffer and handle overhead
  - Multiple RDMA inputs and outputs among four Tofu interfaces are handled by single MPI thread.

Design: Bcast(1)

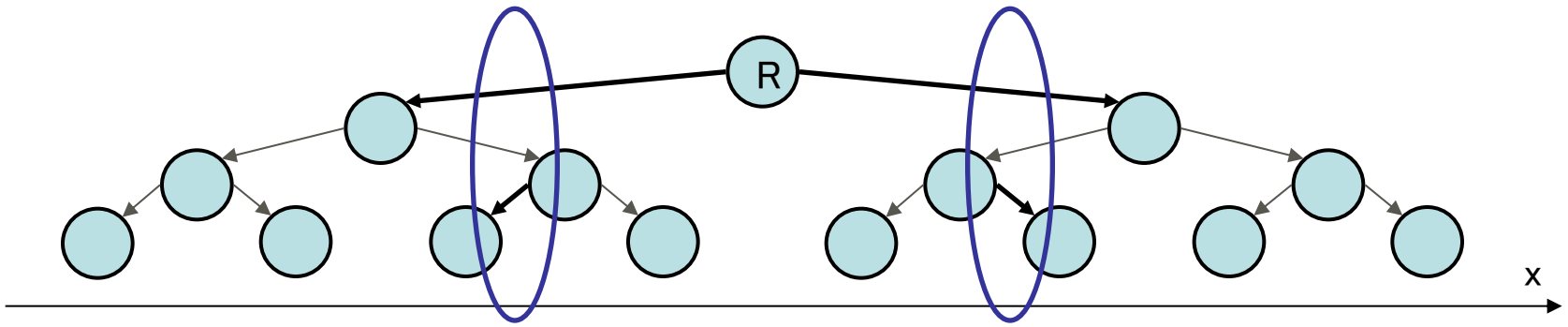
- Long-message algorithm: “Trinaryx3”
  - Communicating along 3 edge-disjoint spanning trees embedded into 3D torus
  - The message is divided into three parts
  - #step = O(X+Y+Z)
    - ~100 steps for >10,000 nodes



Design: Bcast(2)

FUJITSU

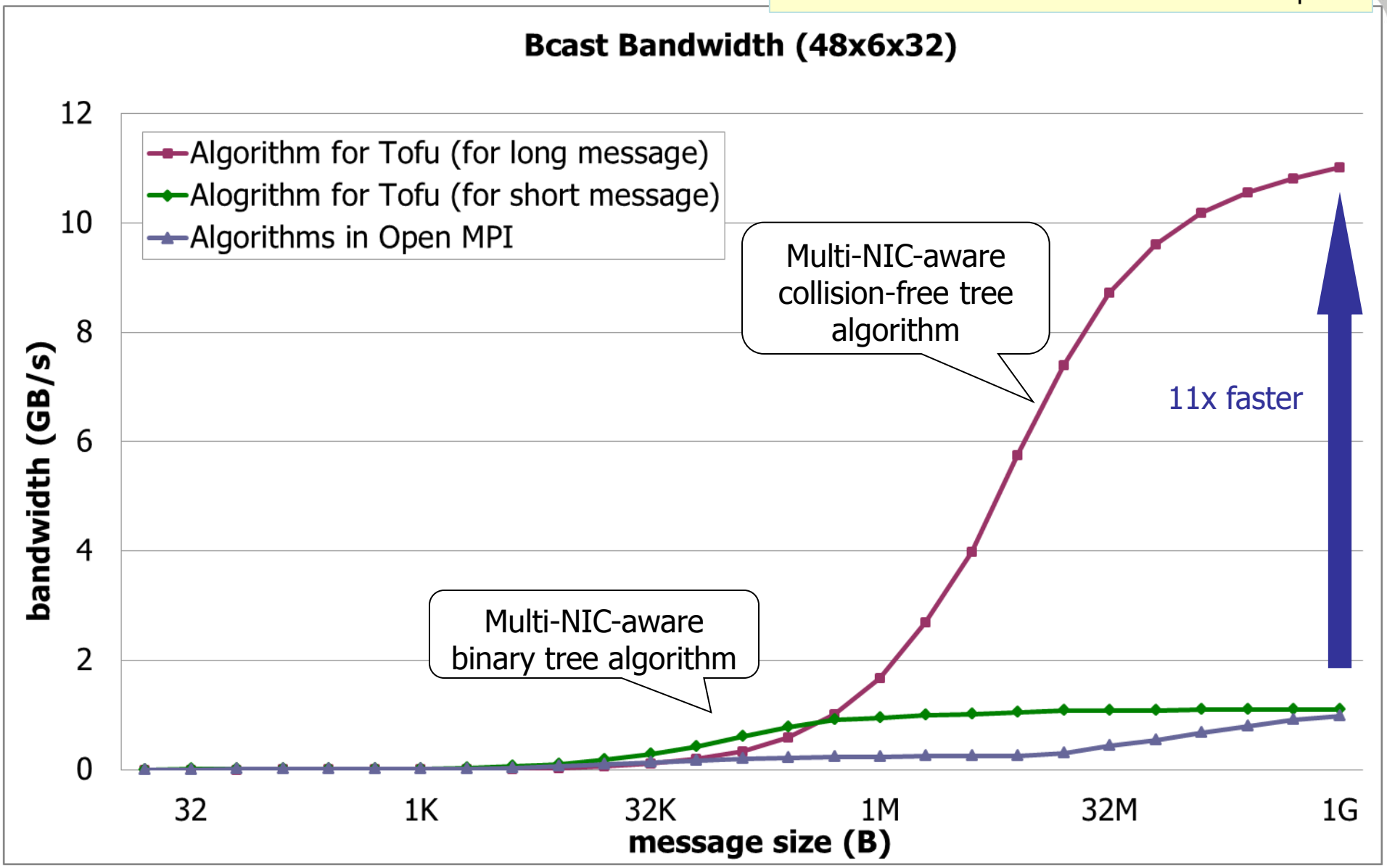
- Short-message algorithm: “3D-bintree”
  - Popular binary tree algorithm, but topology-aware
  - Constructing a binary tree along each axis
    - Some of the edges share the same link (see below)
  - #step = O(logP)
    - ~15 steps for >10,000 nodes



Effective of Collectives: Bcast Bandwidth

FUJITSU

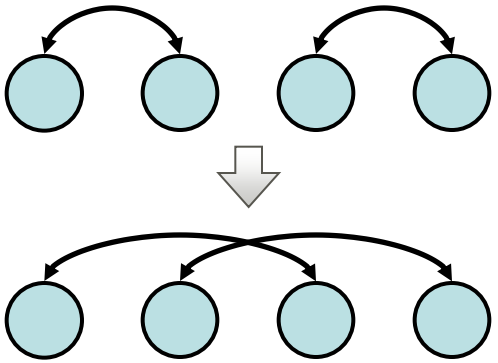
Collaborative work with RIKEN on K computer



Design: Allreduce

FUJITSU

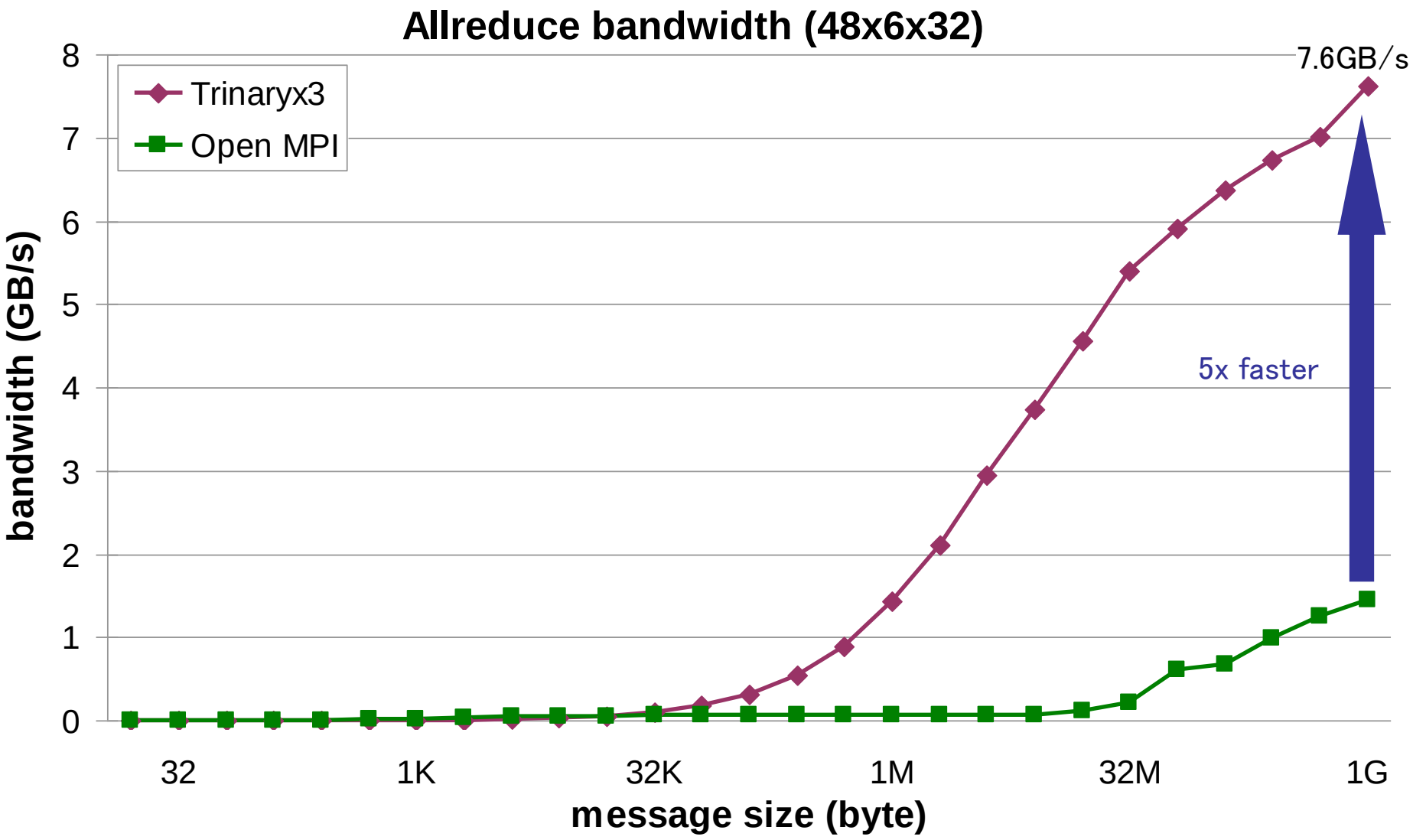
- Long-message algorithm: “Trinaryx3”
  - Trinaryx3 Reduce + Trinaryx3 Bcast
    - Trinaryx3 Reduce can be naturally derived from Trinaryx3 Bcast
    - No overlap between them because there are only 4 TNIs
  - #step = O(X+Y+Z)
    - ~200 steps for >10,000 nodes
- Short-message algorithm: “recursive doubling”
  - Traditional rank-based algorithm
  - #step = O(logP)
    - ~15 steps for >10,000 nodes



Evaluation of Collectives: Allreduce

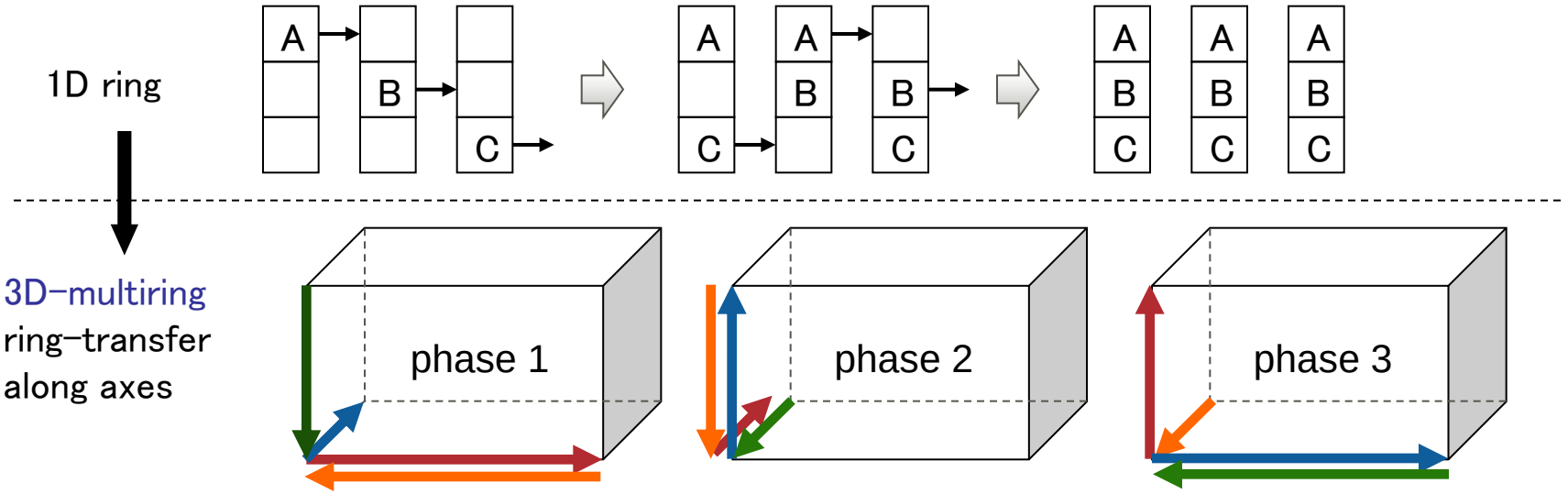
FUJITSU

Collaborative work with RIKEN on K computer



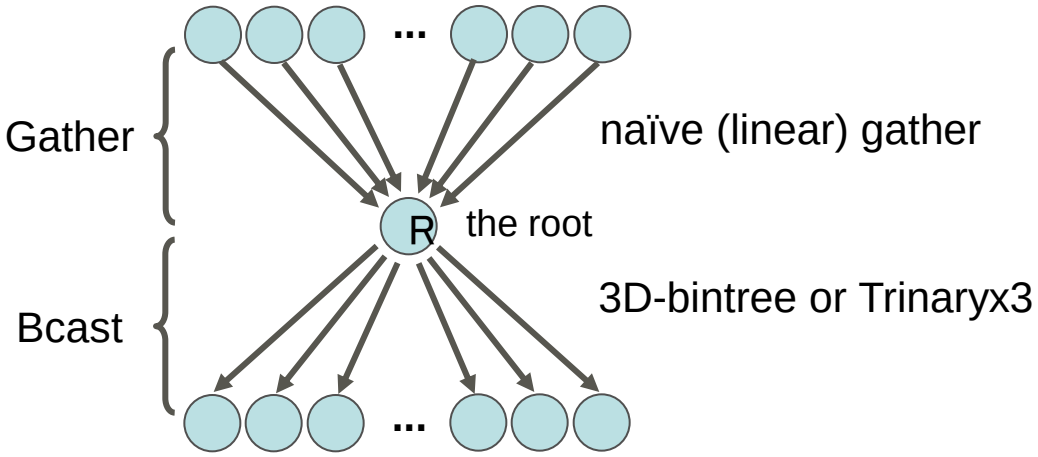
Design: Allgather(1)

- Long-message algorithm: “3D-multiring”
  - Multipath ring-based algorithm
  - The message is divided into (up to) 4 parts
  - Communication directions are chosen such that the 4 streams do not share links at the same time
    - No resource contention will occur in most cases



Design: Allgather(2)

- Short-message algorithm: “Simple-RDMA”
  - Gather + Bcast
  - Gather: naïve direct transfer to the “root” (rank0)
    - The bottleneck will be incoming bandwidth of the root node and the impact of message collisions is small
  - Bcast: Tofu-optimized Bcast (shown before)



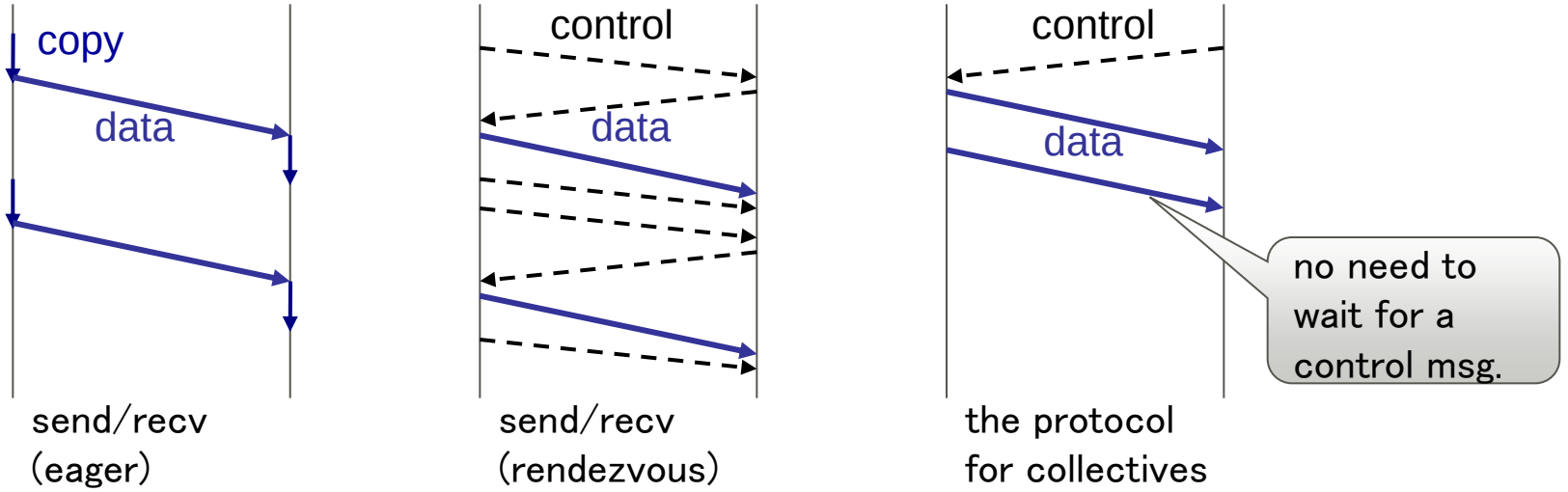
Implementation: Protocol for Collectives

FUJITSU

- Widely used MPI implements collectives with send/receive 2-sided functions due to portability
  - Software overhead: data copies, control messages, etc.

➔Using 1-sided Tofu RDMA APIs to reduce latency

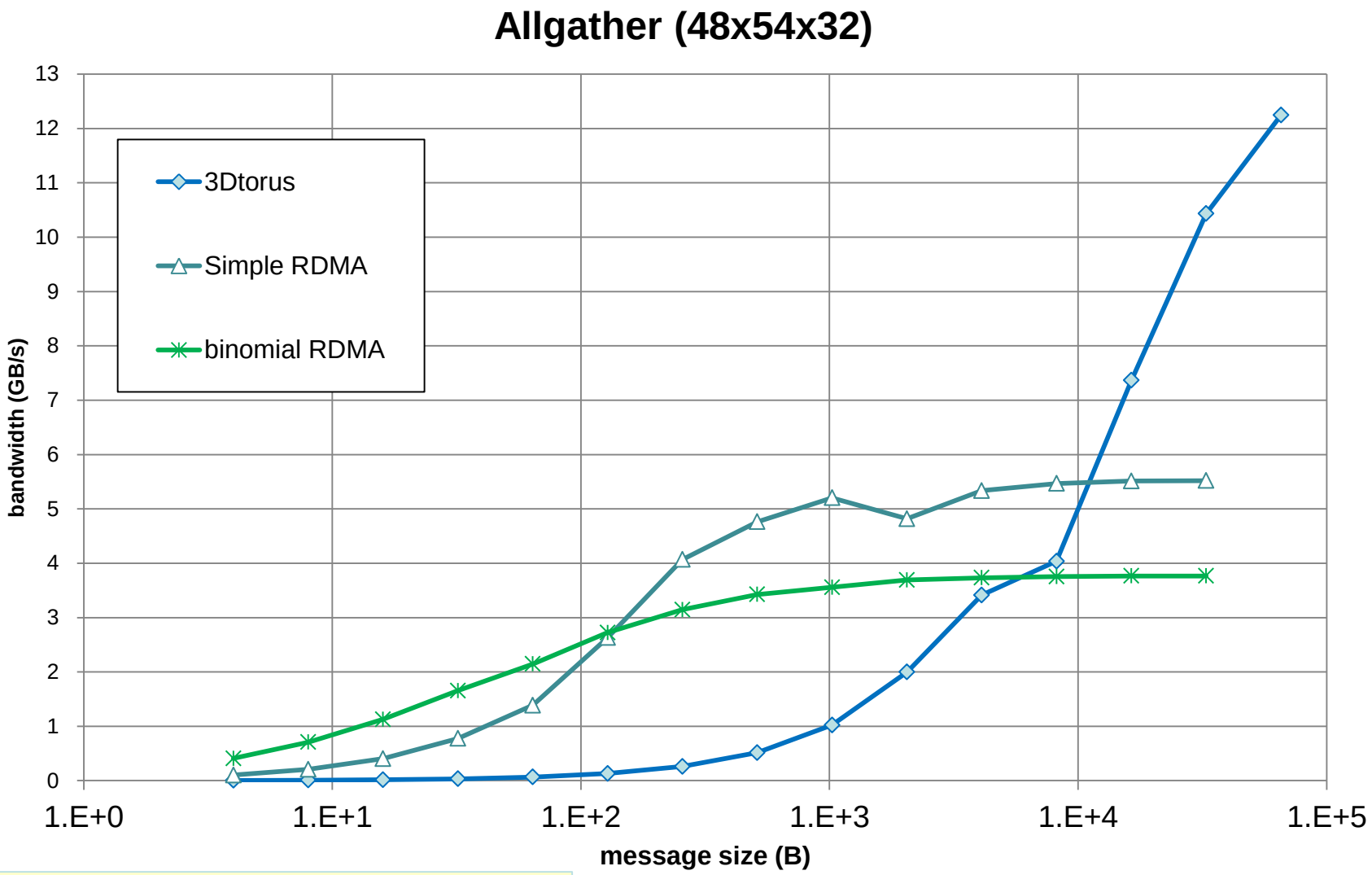
- Comparison of the protocols in pipelined transfer



Evaluation of Collectives: Allgather

FUJITSU

Well Scalability at Full System Scale

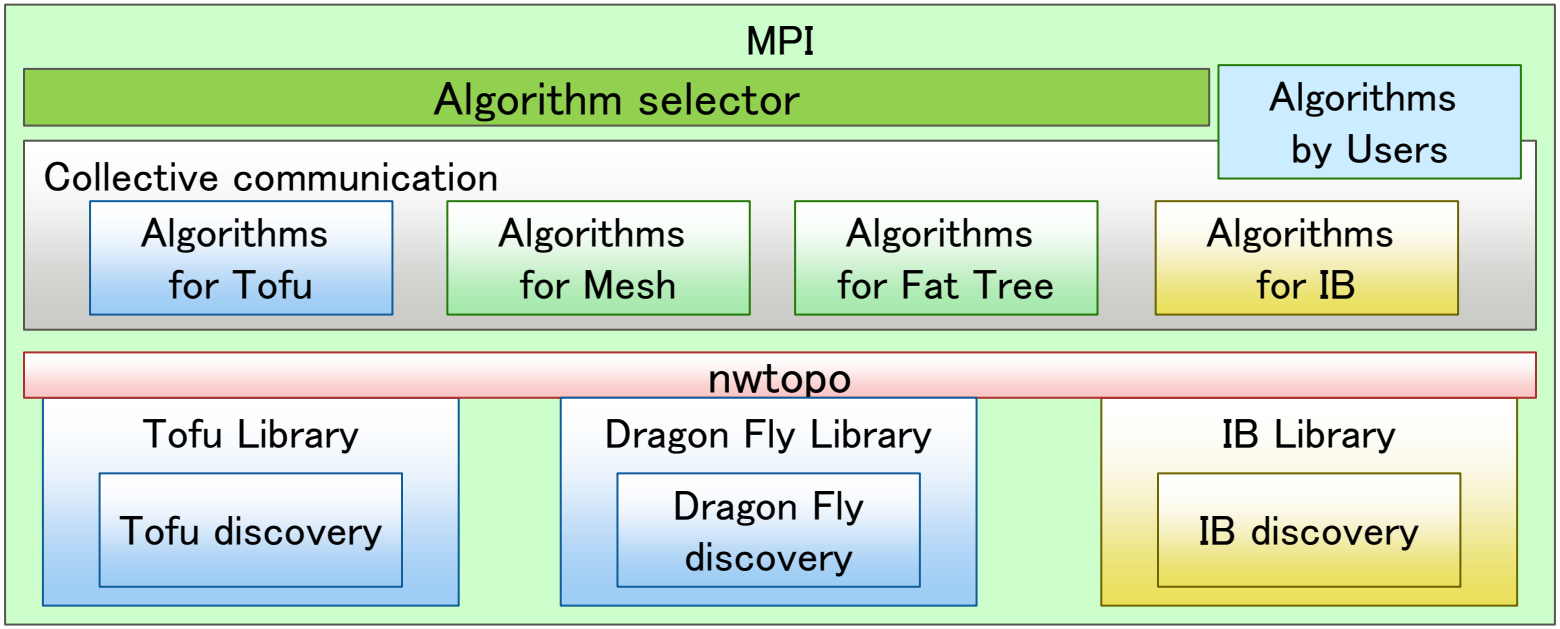


Collaborative work with RIKEN on K computer

Selectable and Tunable Collective Communication Algorithms

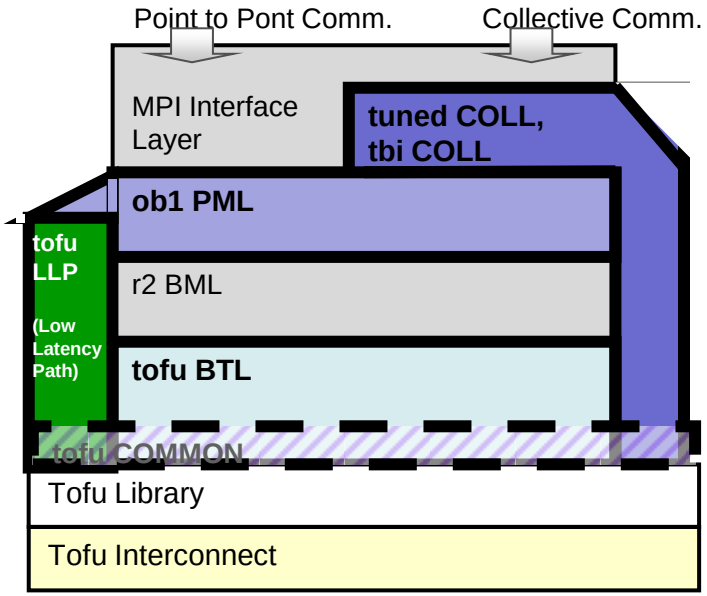
FUJITSU

- Collective Communication algorithms should be selectable and tunable
  - Ex: by message size, by number of nodes, by number of hops.
  - Ex: by 3D dimensional shape,
- Combinations of several algorithms are needed

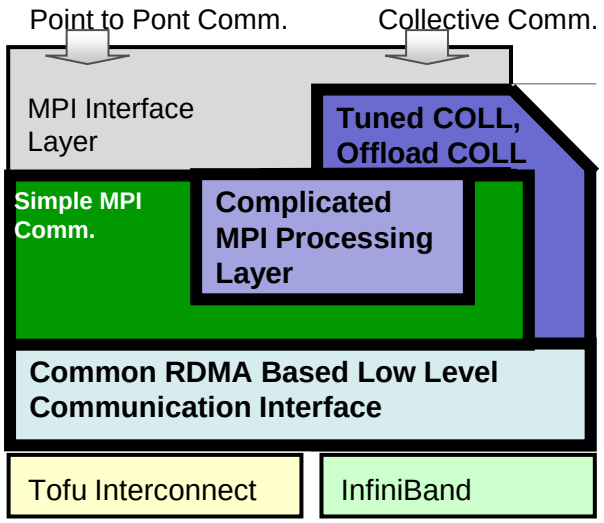


Future MPI Architecture

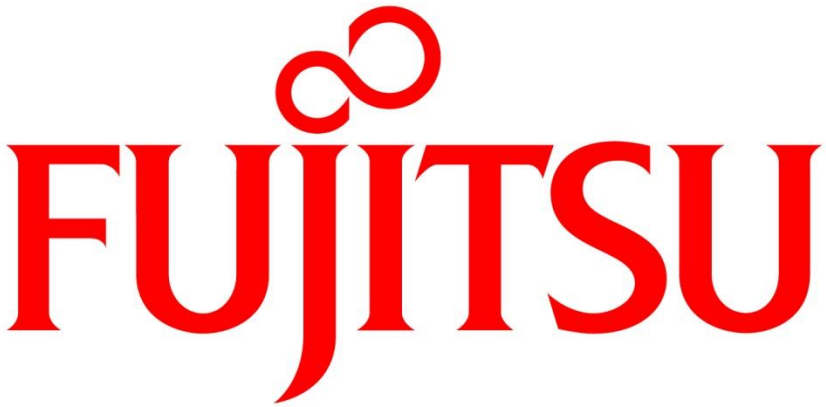
- Current implementation of MPI libraries are message based stack.
  - It is good for portability of collective communication implementation and supporting several environment such as multi-rail
- However the architecture does not fit to RDMA base interconnect such as Tofu interconnect.
- Therefore, future MPI architecture should be simply organized and based on RDMA model.
  - Less Number of Stacks and Converting message passing model to RDMA model



Current Fujitsu MPI Architecture



Future RDMA based MPI Architecture



shaping tomorrow with you

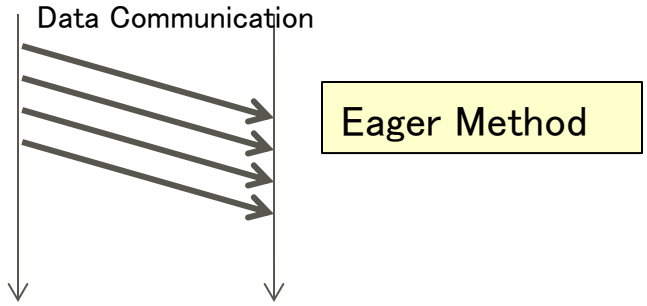
FUJITSU

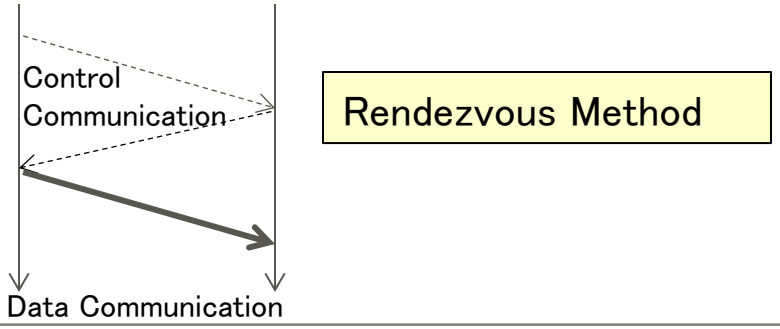
Point to Point Communication Design

Basic Knowledge for High Performance Message Communication Method

FUJITSU

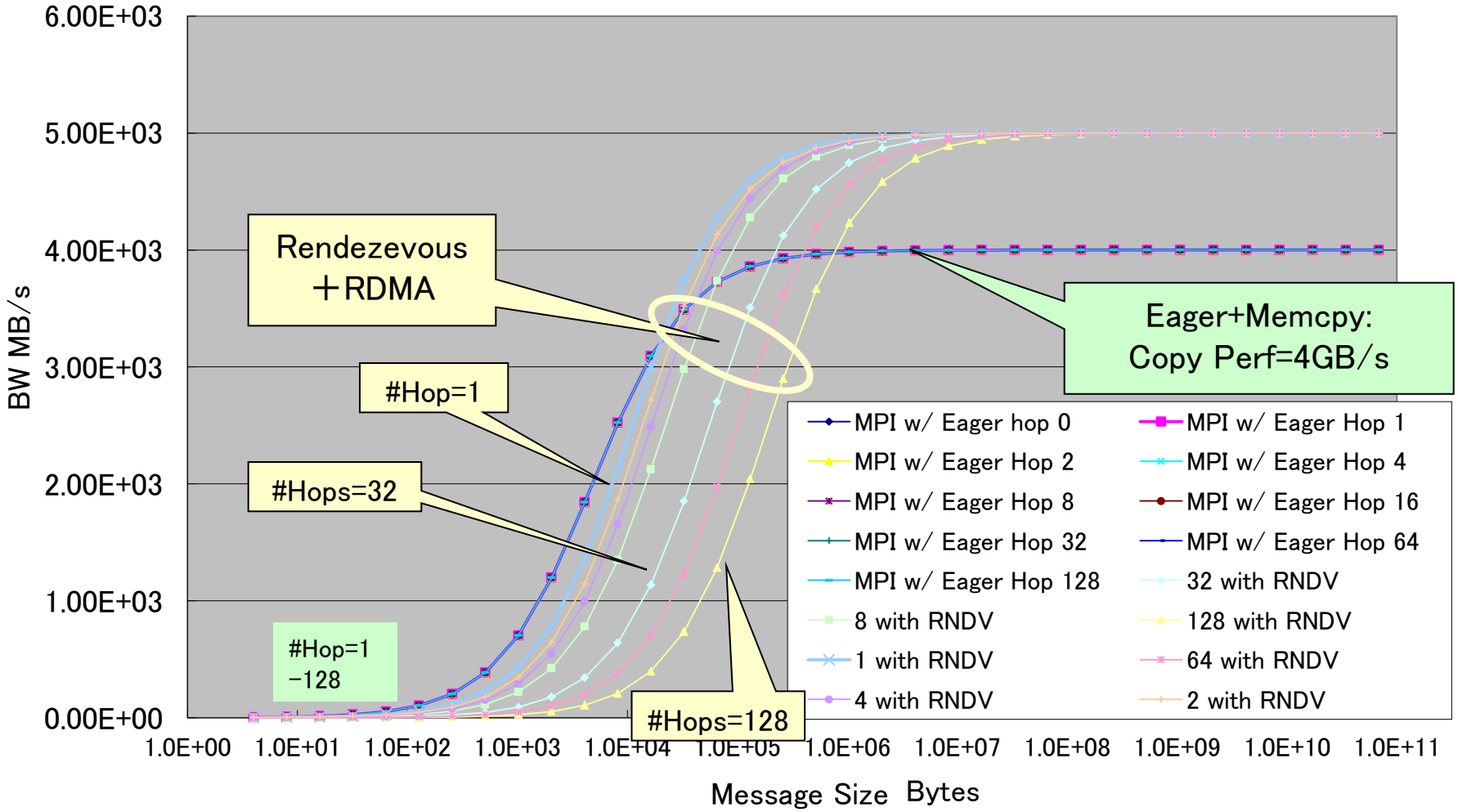
- Eager Method：
  - Pros：Low Latency, High Bandwidth at Short Messages, but performance depends CPU memory copy performance
  - Cons: Requiring receive buffer as much as possible and buffer copy using CPU core（Foreground Communication）
- Rendezvous Method：
  - Pros: No Extra Receive Buffer, Higher Bandwidth with RDMA Data Transfer. No need for CPU Core Processing at RDMA data transfer（Background Communication）
  - Cons: Communication Performance depend on RTT for short message.(Latency Conscious)





Eager vs. Rendezvous Protocol Estimation

FUJITSU

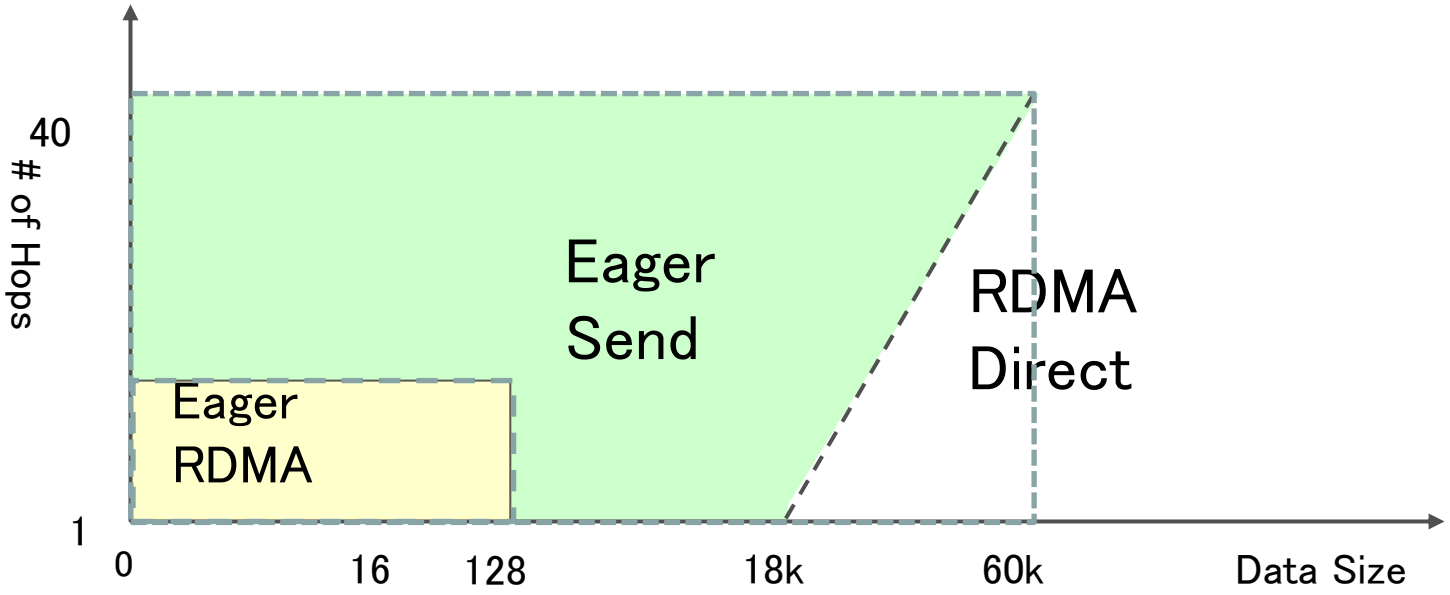


Design Policy: Changing Communication Protocols by #Hops and Message Size

Protocol Selecting Policy for K computer MPI

FUJITSU

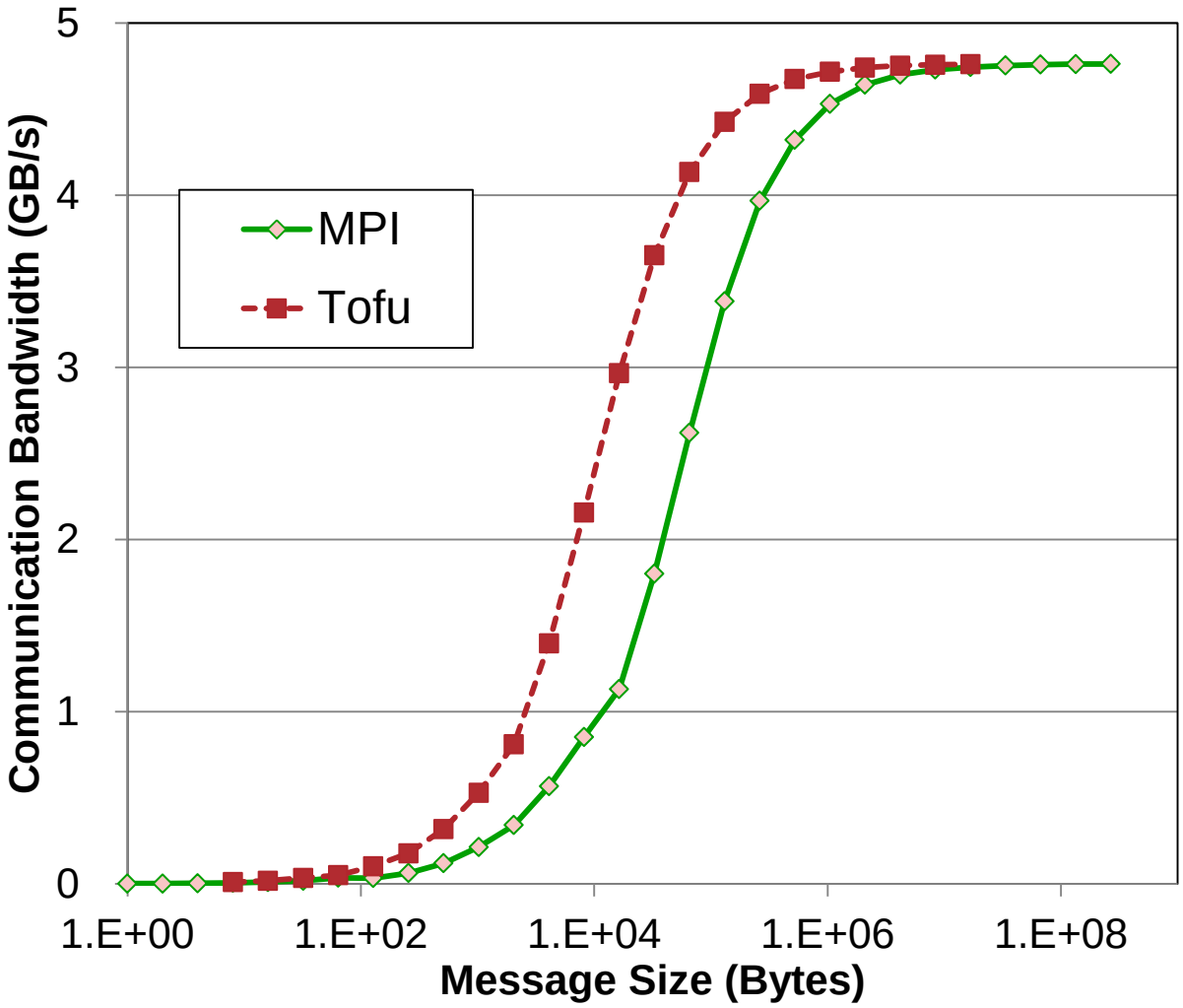
- Changing Communication Protocols by # of Hops and Message Size for Higher Communication Performance



MPI Point to Point Communication Performance of Fast Mode

FUJITSU

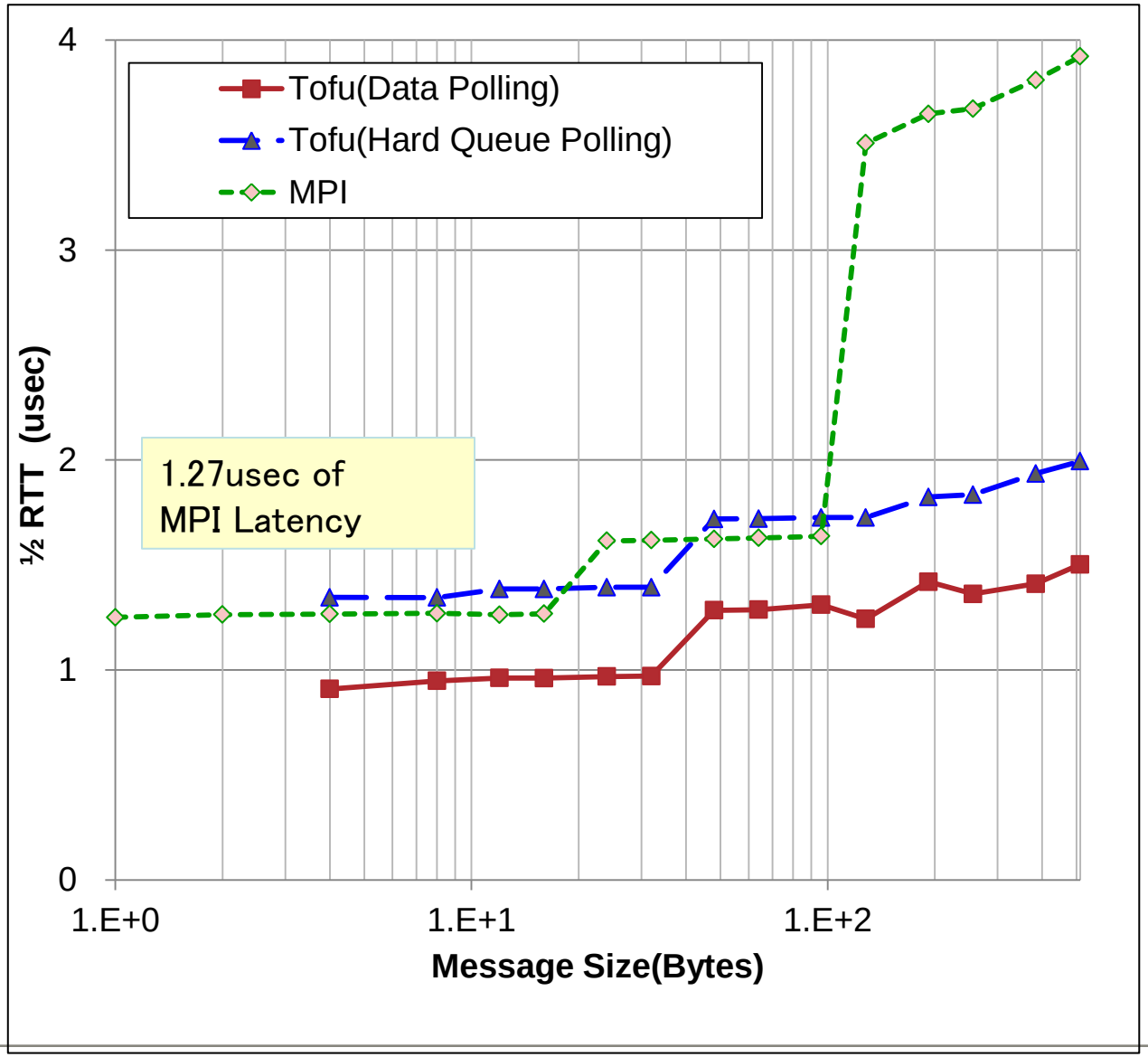
- Realizing smooth protocol connection



MPI vs. Tofu Point to Point Communication

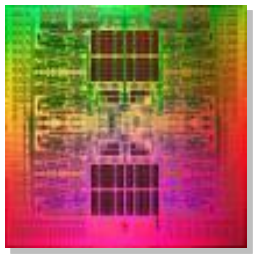
FUJITSU

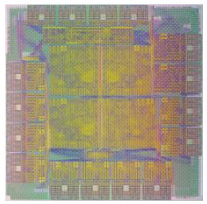
- Software Overhead of MPI is < 300ns

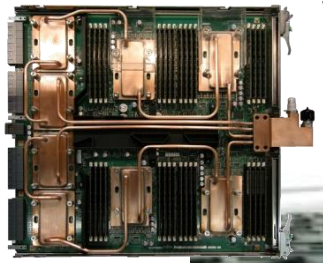


System Overview of K computer

FUJITSU







Processor: SPARC64™ VIIIfx

- Fujitsu's 45nm technology
- 8 Core, 6MB Cache Memory and MAC on Single Chip
- High Performance and High Reliability with Low Power Consumption

Interconnect Controller:ICC

- 6 dims-Torus/mesh (Tofu Interconnect)

System Board: High Efficient Cooling

- With 4 Computing Nodes
- Water Cooling: Processors, ICCs etc
- Increasing component lifetime and reducing electric leak current by low temperature water cooling

Rack:High Density

- 102 Nodes on Single Rack
  - 24 System Boards
  - 6 IO System Boards
  - System Disk
  - Power Units

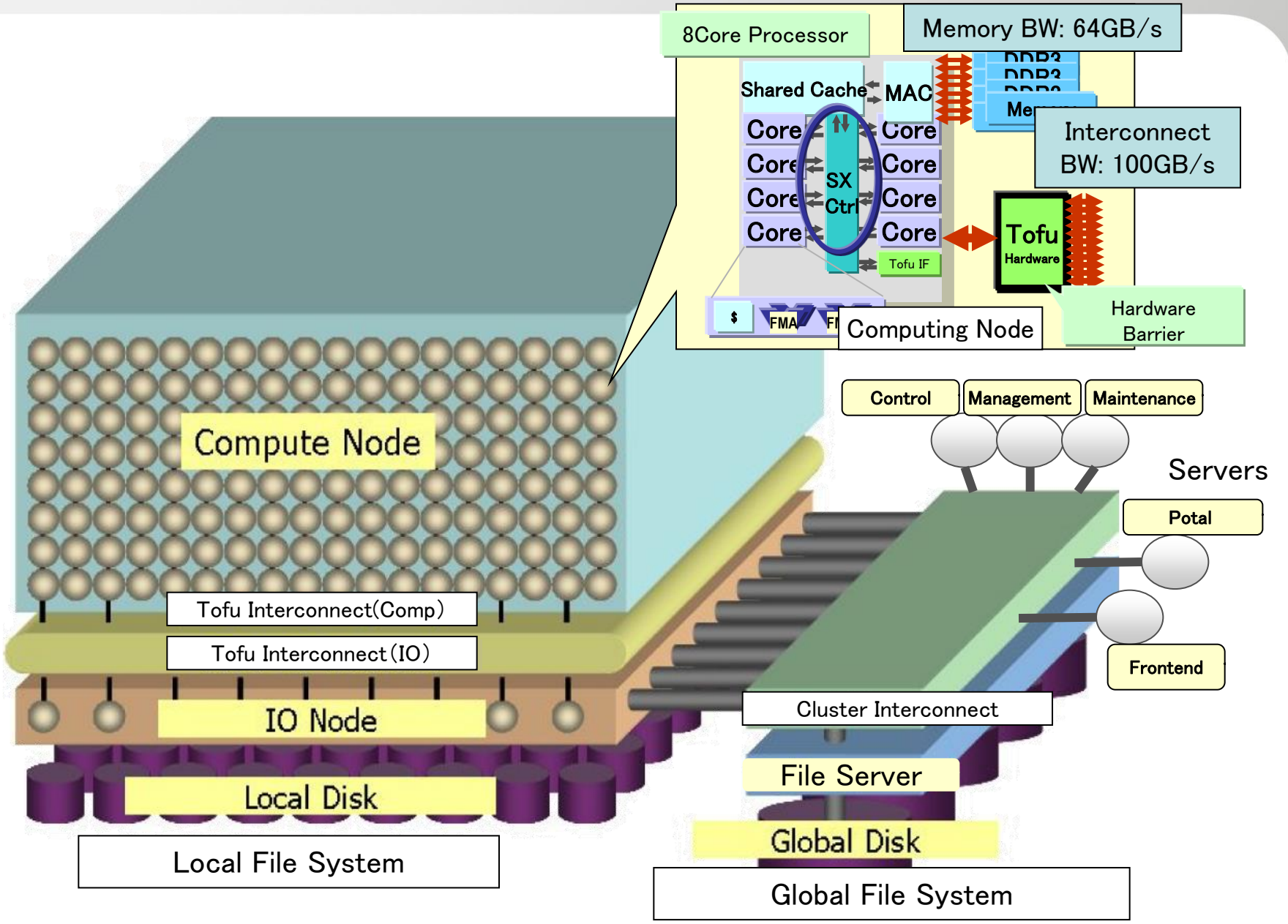
(10PFlops: 864 Racks)

Our Goals

- Challenging to Realize World's Top 1 Performance
- Keeping Stable System Operation over 80K Node System

System Configuration

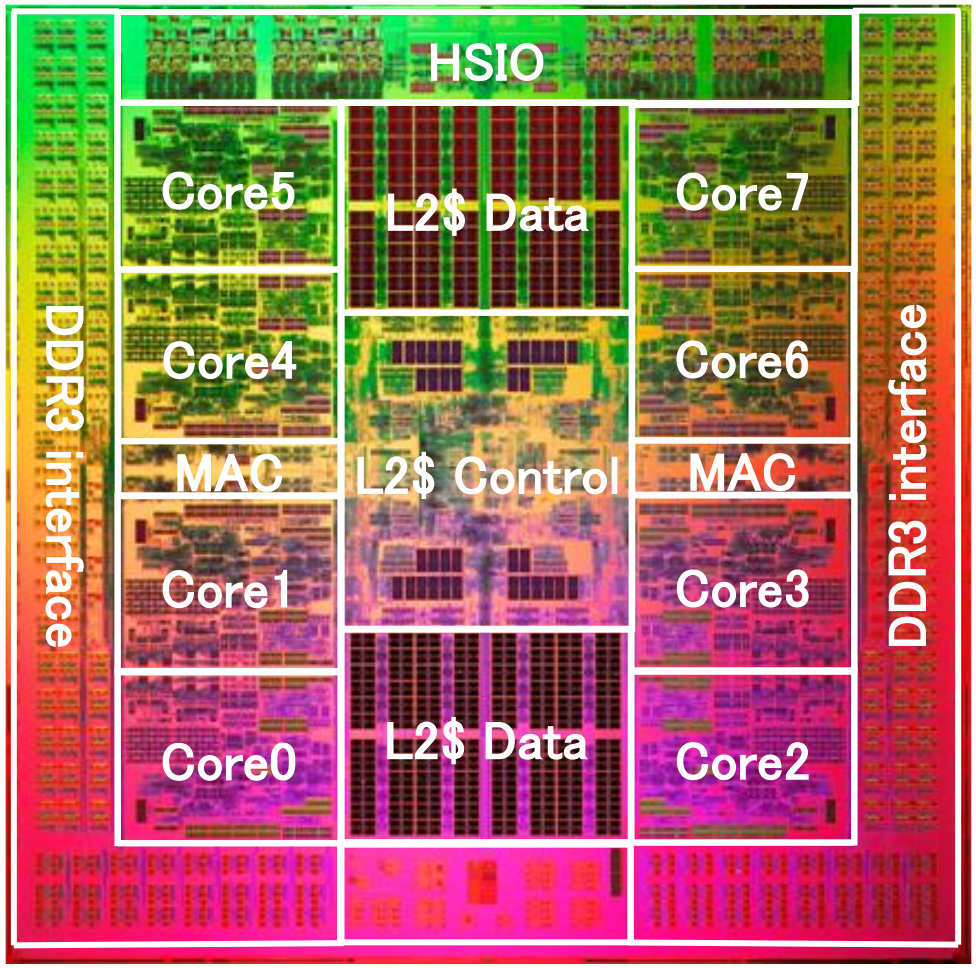
FUJITSU



SPARC64™ VIIIfx Chip Overview

FUJITSU

Design Goals: High Performance and High Reliability with Low Power Consumption

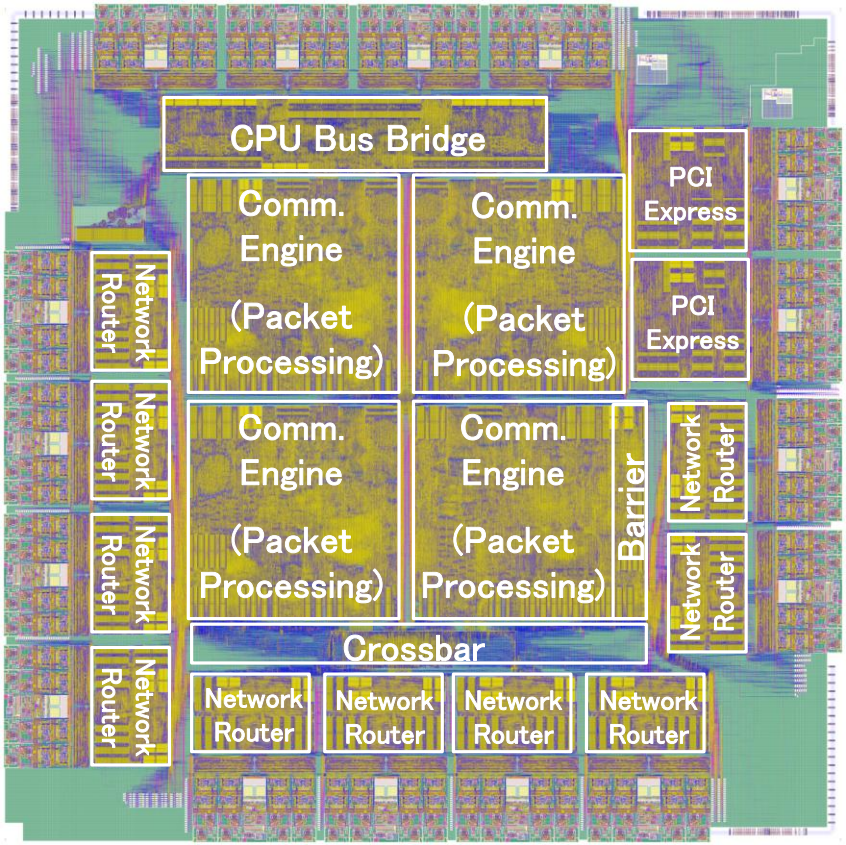


- **Basic Specification**
  - 8 Core, 6MB Shared L2 Cache
  - 8ch DDR3 DIMM MAC
  - Operating Clock 2 GHz
  - HPC-ACE(HPC Extension)
- **FSL 45nm CMOS**
  - 22.7mm x 22.6mm
  - 760M Transistors
  - Signal Pins 1271
- **Peak Performance: 0.5BF**
  - FP Performance 128GFlops
  - Memory Bandwidth 64GB/s
- **Power Consumption**
  - 58W (TYP, 30℃)
  - Water Cooling for Reduction of Leak Current and Increasing Reliability

Tofu: InterConnect Controller(ICC) Chip Design

FUJITSU

Design Goals: **High Bandwidth, Low Latency, High Reliability with Low Power Consumption**



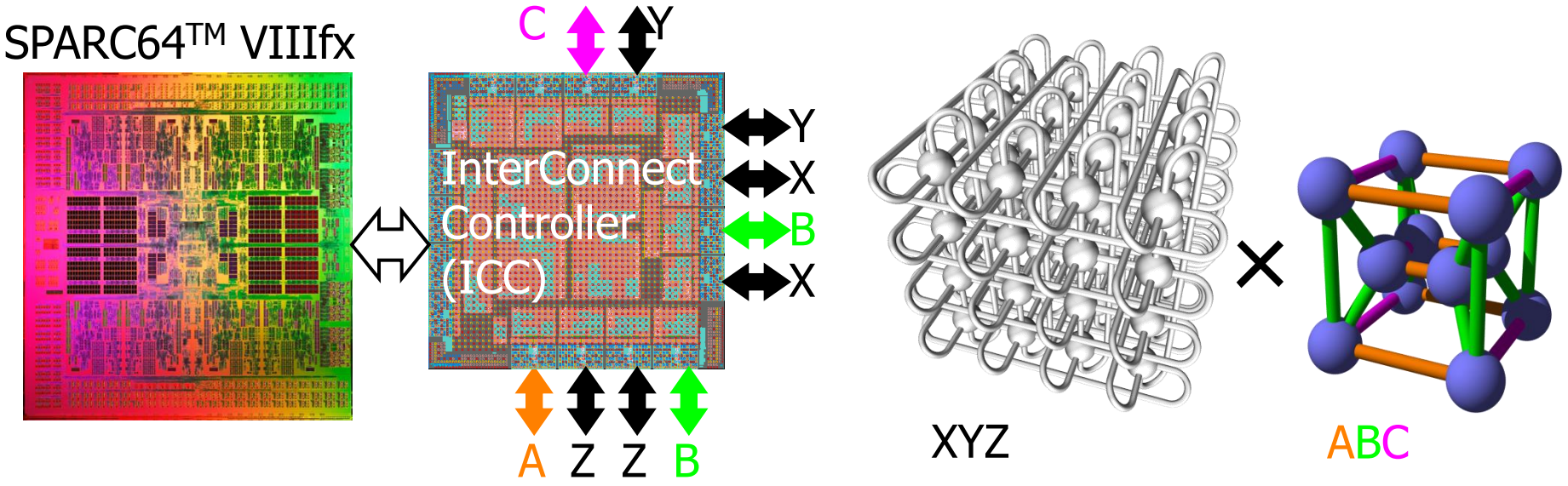
- **ICC Archtecture**
  - RDMA Comm Engine × 4
    Hardware Barrier + Allreduce
  - Network Dimension: 10
  - PCI Express for IO
  - Operating Clock 312.5 MHz
- **FSL 65nm ASIC**
  - 18.2mm x 18.1mm
  - # of gates 48M
  - SRAM 12Mbit
  - # of HSIO: 128 lanes
- **High Bandwidth: 0.31BF**
  - Link Bandwidth 5GB/s × 2
  - Switching Bandwidth 100GB/s (140GB/s, w/ 4 NICs)
- Low Latency
  - Virtual Cut-Through Transfer ~100ns
- Power Consumption
  - 28W (TYP, 30℃) using Water Cooling

The Tofu Interconnect

FUJITSU

- Proprietary Interconnect for SPARC64™ VIIIfx, IXfx
- "**To**rus **fu**sion" 3D-Torus × 3D-Torus = 6D-Torus

Topology	6D Torus/Mesh
Coordinate Axes	X, Y, Z, A, B, C
Max. Network Size	32, 32, 32, 2, 3, 2
System Configuration of K computer	Torus: X, Z, B / Mesh: Y, A, C Computing Nodes: Z = 1～16 ／ IO Nodes: Z = 0



Why Introducing 6D Torus/Mesh Interconnect

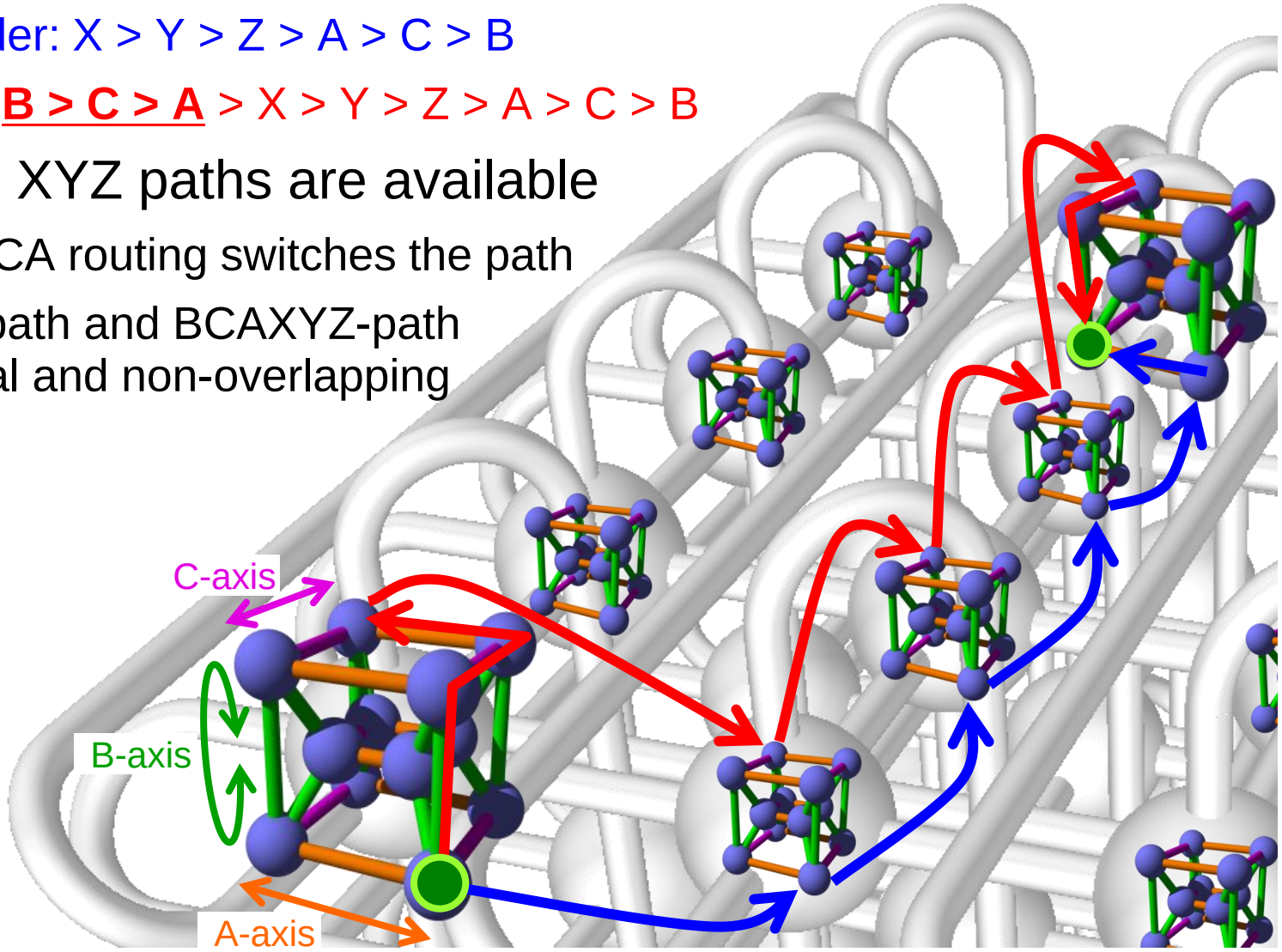
FUJITSU

- Reduction of Latency (Total Number of Hops)
  - Average # of Hops: ½ of 3D Torus
- Increasing Bisection Bandwidth
  - 1.5 times better than 3D torus
- Fault Tolerance
  - 12 way Software Controlled Routing
- For Easy to Use
  - Building 3D Torus Cube by combining two of 6D axis
  - User does not recognize 6D interconnect

Routing Algorithm

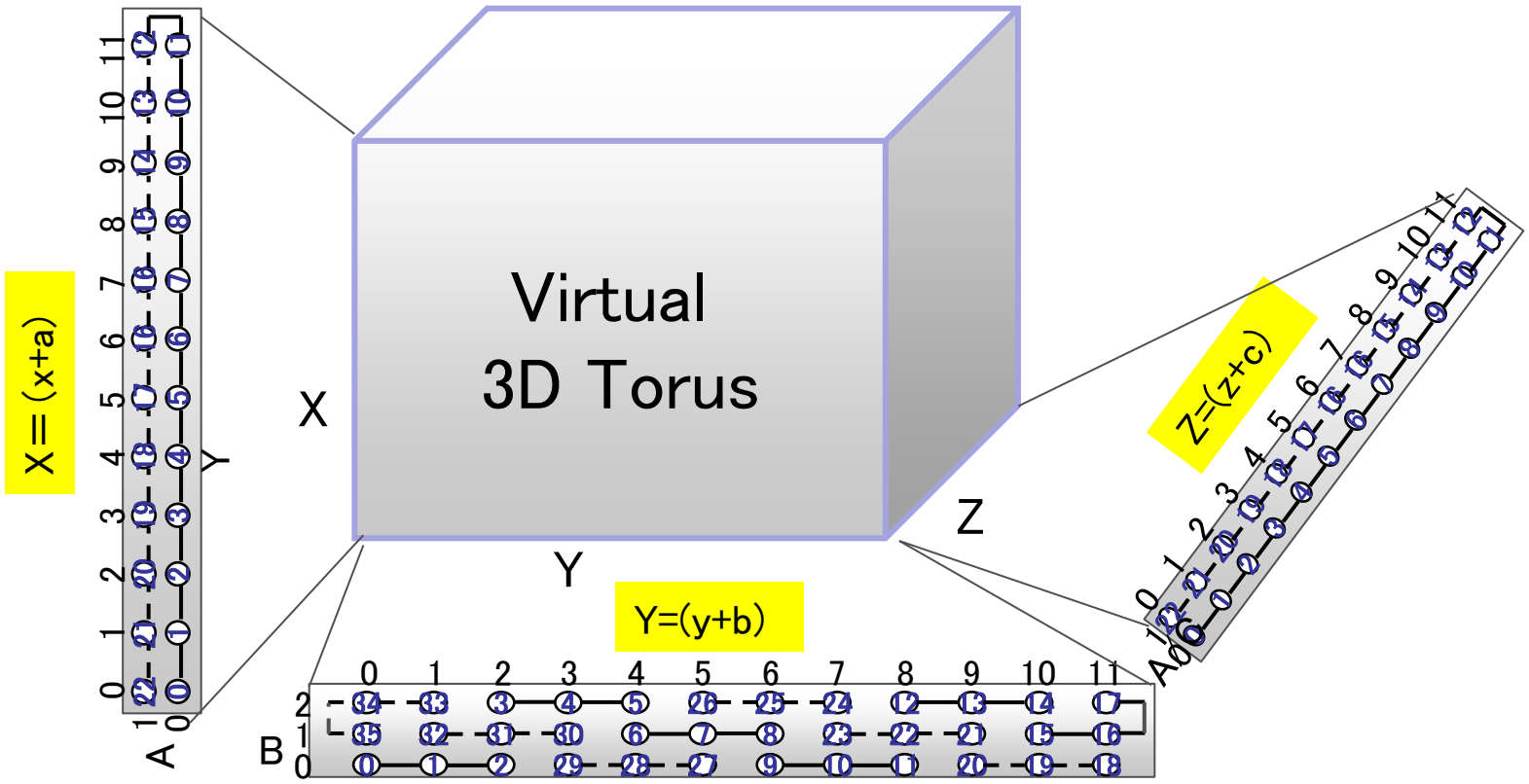
FUJITSU

- Extended dimension-order algorithm
  - Default order: X > Y > Z > A > C > B
  - Extended: **B > C > A** > X > Y > Z > A > C > B
- 3x2x2 = 12 XYZ paths are available
  - The first BCA routing switches the path
  - XYZACB-path and BCAXYZ-path are minimal and non-overlapping



Job Allocation and Rank Mapping on Tofu

- User can specify 1,2,3D network on job submission.
- Job scheduler makes torus network by combination of XYZ+ABC
  - User can specify the combination
- Basic Combination of 3D Torus Mapping: X=x+a, Y=y+b, Z=z+c



Programming Interface of Tofu RDMA Engines

FUJITSU

- Tofu RDMA Engine Provides:
  - Simple RDMA Interface: Put, Get, Barrier, Allreduce (Reduce+Broadcast)
  - 4 RDMA Engines provide 40GB/s bi-directional BW
  - Simple Page Tables to bind physical memory (STAG)
  - Providing Scalable Communication: not Connection Oriented
    Able to start RDMA by only setting STAG.