Chapter 9
MPI and Inter-node Tuning

FUJITSU LIMITED
April 2016

Contents

Outline and explanation part

- MPI Specifications of the FX100
- Compilation and Execution of an MPI Program

MPI Specifications of the FX100

- The MPI-3.0 standard is supported.

  The following sections in the MPI standard correspond to the supported MPI-3.0 standard. (For details, see the *MPI User's Guide*.)

  - 3.8.2 Matching Probe
  - 3.8.3 Matched Receives
  - 5.12 Nonblocking Collective Operations

  * The MPI-3.0 standard is supported in V2.0L30.

- Thread level
  - MPI_THREAD_SERIALIZED is supported.

- Specifications have been extended from the MPI standard.
  - Rank Query Interface
  - Extended RDMA interface
  - Section specifying MPI statistical information interface
  - Extended Persistent Communication Requests Interface
  - MPI Asynchronous Communication Promotion Section Specifying Interface

Compilation and Execution of an MPI Program

- Compilation of an MPI program
  - How to compile the program
  - MPI program compile options
- Execution of an MPI Program
  - How to execute the program
  - mpiexec options (global options)
  - mpiexec options (local options)
  - MCA parameters
- SPMD model and MPMD model

How to Compile the Program

Compilation of an MPI program

- Use the following commands to compile an MPI program.

Source program	Command (cross)	Command (own)
Fortran	mpifrtpx	mpifrt
C	mpifccpx	mpifcc
C++	mpiFCCpx	mpiFCC

- Optimization options are the same as those of Fortran, C, and C++ compilers.
- There are three types of MPI-specific options as follows.

Option	Meaning
--showme [:{compile|link|version}]	Displays compile command, link command, and version information.
-SCALAPACK	Links the ScaLAPACK library.
-SSL2MPI	Links the SSL II/MPI library.

  - To use -SCALAPACK or -SSL2MPI, also specify the -SSL2 or -SSL2BLAMP option.

How to Execute the Program (1/2)

FUJITSU

- Job execution options (MPI-related pjsub options)
  - -L node={X|XxY|XxYxZ}
    - This option specifies the number of nodes required for the entire job.
    - The system secures the specified one to three dimensional torus shape.
  - --mpi shape={X|XxY|XxYxZ}
    - This option specifies the shape of MPI_COMM_WORLD. Specify this option to execute dynamic process generation.
    - If omitted, the setting is the same as 'node'.
  - --mpi proc=N
    - This option specifies the size of MPI_COMM_WORLD.
    - If omitted, the setting will be the product of 'node' (e.g. XxY or XxYxZ), or the product of 'shape' (e.g. XxY or XxYxZ) if 'shape' is specified.

How to Execute the Program (2/2)

- Job execution options (MPI-related pjsub options)
  - --mpi {rank-map-bynode|rank-map-bychip[:rankmap]|rank-map-hostfile=filename}
    - This option specifies the rank assignment rule for generated processes.
    - rank-map-bynode assigns one process to a node. After assignment has been done for all nodes, it returns to the first node that was assigned a process.
    - rank-map-bychip assigns the number of processes specified by 'rankmap' to a node and then proceeds to the next node.
    - rank-map-hostfile assigns ranks to generated processes according to the specified 'filename' file.
    - If omitted, the setting is the same as rank-map-bychip.

mpiexec Options (Global Options) (1/2)

FUJITSU

Global options : Options affecting all of mpiexec

Option	Meaning
{-debuglib|--debuglib}	Uses a debug library.
{-h|--help}	Displays help messages.
{-of|--of|-std|--std} OF_FILE	Outputs the standard output and standard error output to OF_FILE.
{-oferr|--oferr|-stderr|--stderr} OFERR_FILE	Outputs the standard error output to OFERR_FILE.
{-oferr-proc|--oferr-proc|-stderr-proc|--stderr-proc} OFERR_PROC_FILE	Outputs the standard error output to OFERR_PROC_FILE for each process.
{-ofout|--ofout|-stdout|--stdout} OFOUT_FILE	Outputs the standard output to OFOUT_FILE.
{-ofout-proc|--ofout-proc|-stdout-proc|--stdout-proc} OFOUT_PROC_FILE	Outputs the standard output to OFOUT_PROC_FILE for each process.
{-ofprefix|--ofprefix|-stdprefix|--stdprefix} OFPREFIX	Prefixes an identifier at the beginning of the each line of standard output and standard error output. For OFPREFIX, specify one of {rank|nid|rank,nid|nid,rank}.
{-stdin|--stdin} STDIN_FILE	Specifies the standard input file.
{-app | --app} APP_FILE	Uses the APP_FILE file to specify local options and an executable file.
{-nompi | --nompi}	Uses parallel execution for an executable file that is not an MPI program.

mpiexec Options (Global Options) (2/2)

■ Global options : Options affecting all of mpiexec

Option	Meaning
{-vcoordfile|--vcoordfile} VCOORD_FILE	Uses background execution.
{-V|--version}	Outputs version information.

mpiexec Options (Local Options)

FUJITSU

■ Local options : Options specified for each executable program

Option	Meaning
-am AM_FILE	Specifies the configuration file of MCA parameters.
-x NAME=VALUE	Sets the environment variable NAME with the value of VALUE in an MPI program.
{-mca|--mca} MCA_PARAM_NAME MCA_PARAM_VALUE	Sets the value of MCA_PARAM_VALUE in the MCA parameter MCA_PARAM_NAME.
{-c|-np|--np|-n|--n} N	Specifies the number of parallel processes. For MPMD model, this option cannot be omitted. If omitted for an SPMD model, the setting will be the value of --mpi proc. If --mpi proc is not specified, it will be the product of the --mpi shape element (or the product of -L node if --mpi shape is not specified).

MCA Parameters (Mainly Effective for Point-to-point Communication)

MCA parameter	Meaning
btl_tofu_eager_limit	Changes the threshold for switching between the Eager and Rendezvous communication methods.
common_tofu_fastmode_threshold	Specifies the communication count threshold at which the mode switches from memory-saving communication mode to fast communication mode. The default is 16.
common_tofu_large_recv_buf_size	Changes the size of the Large receive buffer used in fast communication mode. Specify 1024 or more. The default is 1 MiB.
common_tofu_max_fastmode_procs	Specifies the upper limit on the number of processes in fast communication mode. The default is 1024.
common_tofu_max_tnis	Specifies the number of TNIs (networks) used. The maximum number of TNIs that the system can use is already set. You do not need to change the value.
common_tofu_medium_recv_buf_size	Changes the size of the Medium receive buffer. Specify 256 or more. The default is 2 KiB.
common_tofu_memory_limit	Specifies the memory usage for MPI. The unit is MiB.
common_tofu_memory_limit_peers	Specifies the assumed number of communication partner processes when calculating the memory usage for MPI.
common_tofu_packet_gap	Specifies a send gap.
common_tofu_packet_mtu	Specifies the maximum packet transfer size.
common_tofu_use_multi_path	Uses trunking in point-to-point communication. (This is not always faster because it may facilitate communication contention.)

MCA Parameters (Effective for Collective Communication)

MCA parameter	Meaning
coll_base_reduce_commute_safe	Guarantees the order of reduction operations. (This parameter has a significant impact on performance. Do not specify the parameter unless you need extremely high precision.)
coll_tbi_use_on_bcast	Uses Tofu barrier communication with the MPI_Bcast function. The default is to use Tofu barrier communication. Use this parameter if you execute a program that violates the MPI standard.
coll_tuned_prealloc_size	Ensures that collective communication parameters share the work buffer. This parameter is used for some algorithms of Allreduce, Reduce, Reduce_scatter_block, Reduce_scatter, Allgather, Gather, Scatter, and Alltoall. The default is 6 MiB.

MCA Parameters (Other)

FUJITSU

MCA parameter	Meaning
dpm_ple_socket_timeout	Specifies the socket communication wait time used when establishing communication between MPI process groups that do not share a communicator.
mca_base_param_file_prefix	Specifies the AMCA parameter file.
mpi_check_buffer_write	Monitors buffer destruction by the nonblocking send function.
mpi_deadlock_timeout	Specifies the period after which processing stops waiting for communication.
mpi_deadlock_timeout_delay	Specifies the period from message output to actual program end.
mpi_preconnect_mpi	Establishes connections within the MPI_Init function.
mpi_print_stats	Outputs MPI statistical information.
mpi_print_stats_ranks	Specifies the ranks for which MPI statistical information is output.

SPMD model and MPMD model

SPMD

- Execute single program.
- When not specifying the -n option, the default value is used for the number of processes.
- When specifying the -n option, be sure not to exceed the pjsub option --mpi proc=N.

```
mpiexec [global options] [local options] executable program
```

* The default value of option "-n" is the maximum number of parallel processes which can be generated.

MPMD

- Execute multiple programs by delimiting execution units with a colon (:).
- The sum of the number of -n options must not exceed the pjsub option --mpi proc=N.

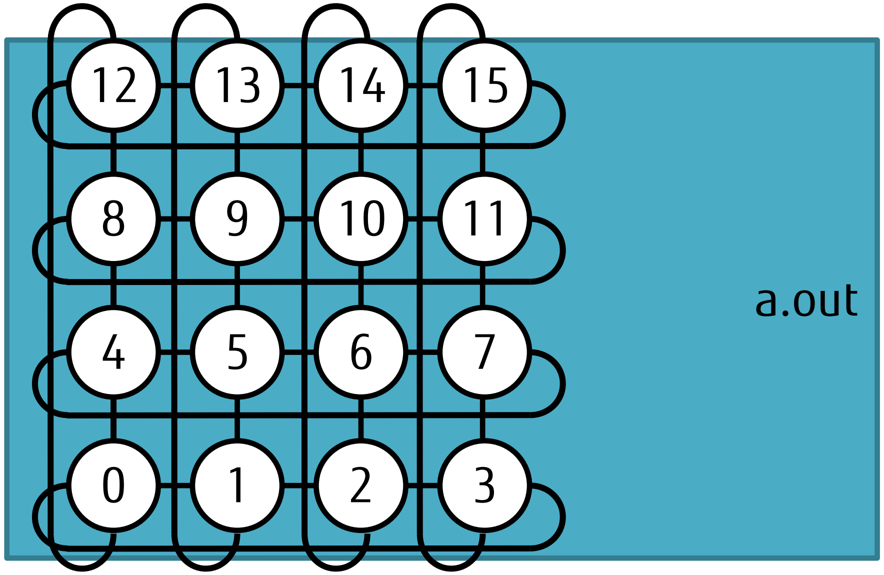
```
mpiexec [global options] ¥
    -n N1 [local options] executable program 1 : ¥
    -n N2 [local options] executable program 2 : ...
```

How to Execute SPMD Model

- The most efficient approach is to have all the nodes in the N-dimensional torus generate processes.

run.sh contents

```
#!/bin/sh
#PJM -L node=4x4
mpiexec ./a.out     # The -n option is not required.
```



The entire 4x4 two-dimensional torus shape has 16 processes assigned to it.

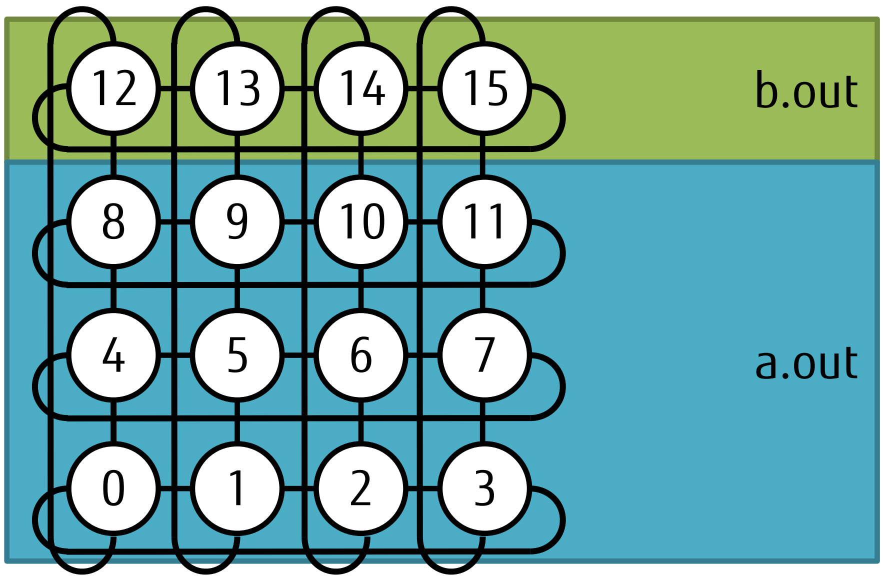
Conceptual Diagram of MPMD Model Process Mapping

FUJITSU

- Ranks are assigned on MPI_COMM_WORLD.

run.sh contents

```
#!/bin/sh
#PJM -L node=4x4
mpiexec -n 12 ./a.out : -n 4 ./b.out
```



In the 4x4 two-dimensional torus,

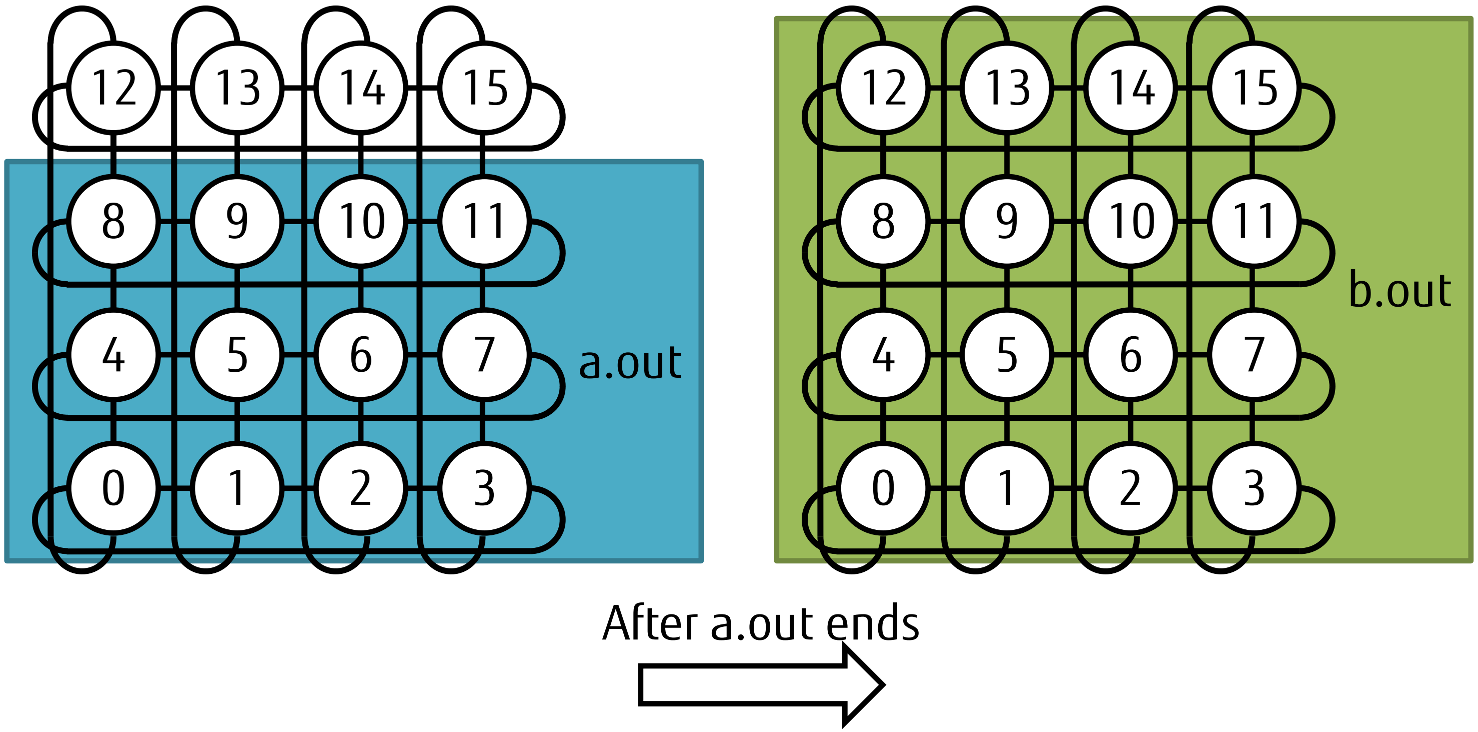
12 processes (0 to 11) are assigned to a.out, and 4 processes (12 to 15) are assigned to b.out.

Execution of Multiple MPI Programs

- Part of the shape specified by 'node' is used.

run.sh contents

```
#!/bin/sh
#PJM -L node=4x4
mpiexec -n 12 ./a.out
mpiexec -n 16 ./b.out
```

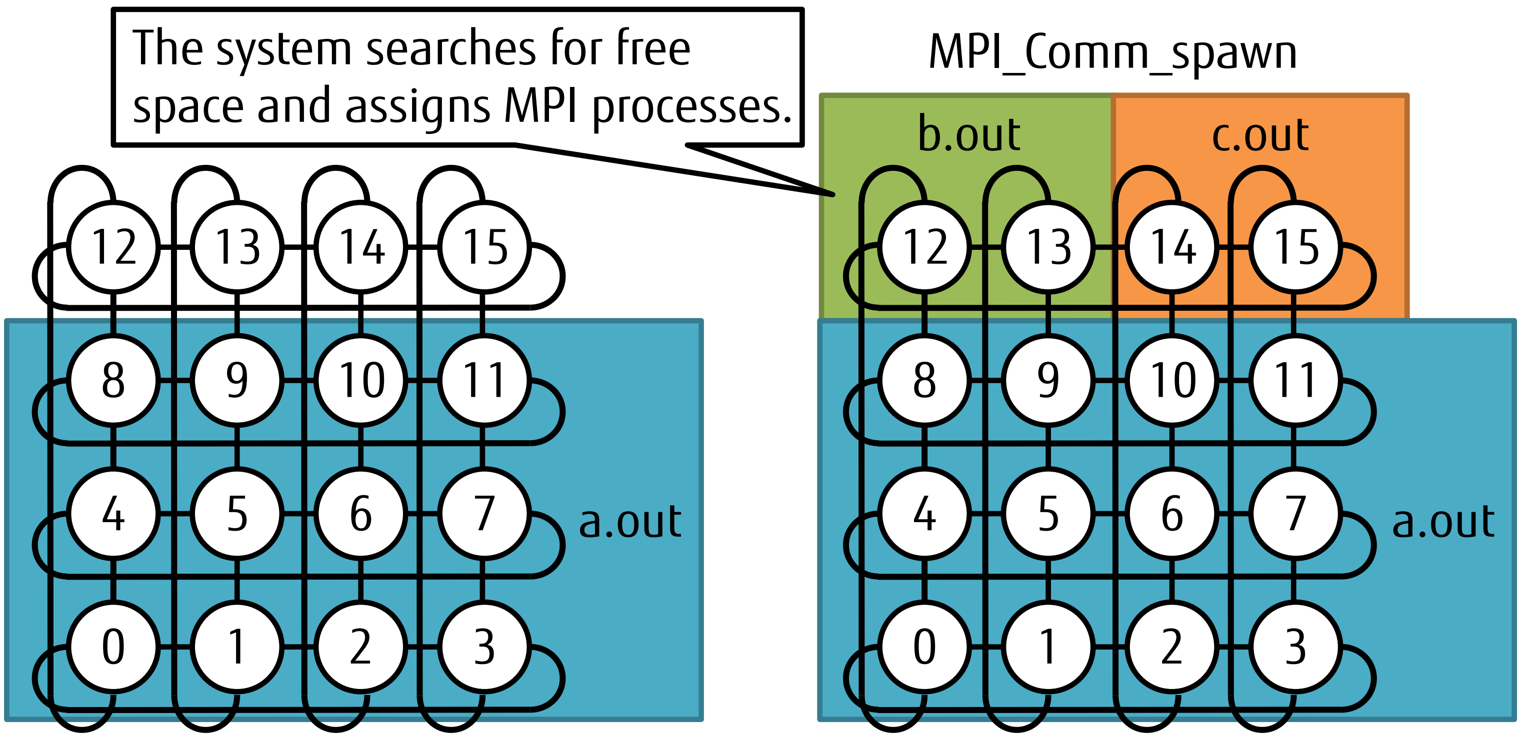


Execution of Dynamic Process Generation

- Prepare space for dynamic processes by using the 'shape' option.

run.sh contents

```
#!/bin/sh
#PJM -L node=4x4
#PJM --mpi shape=4x3
mpiexec  ./a.out
```



Practice and example part

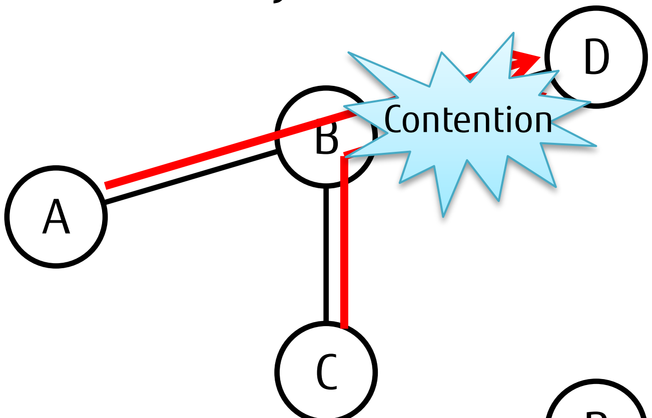
- Performance Improvement
- Tuning Examples
- Fujitsu Extended Specifications
- Troubleshooting

Performance Improvement

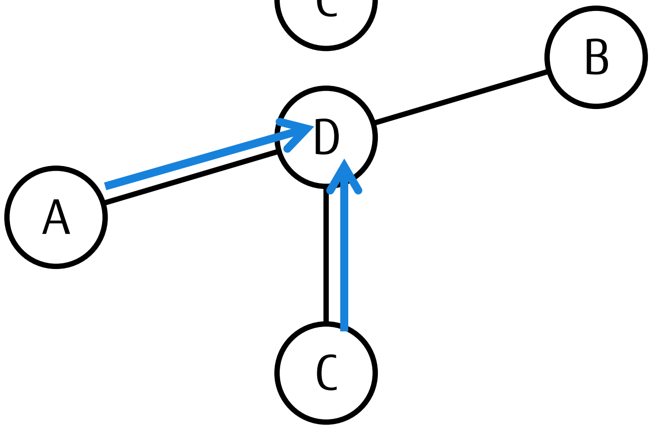
- Communication contention
- Eager and Rendezvous
- Cost of reception wait
- Acceleration of collective communication
- MPI statistical information

Communication Contention

- If there is only adjacent communication, it is not affected by other communication.
  - In communication with discrete communication destinations, contention with other communication may occur.
  - There is no contention where the only form of communication by all members is adjacent communication.



If communication from A to D and communication from C to D are simultaneous, contention occurs between B and D.



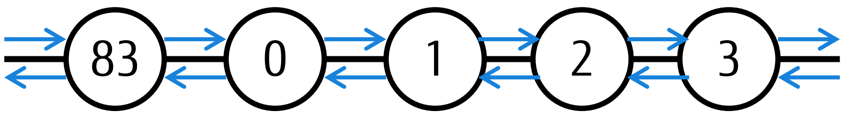
Since communication from A to D and communication from C to D are both adjacent communication, no contention occurs.

FUJITSU

Comparison with IMB exchange performance

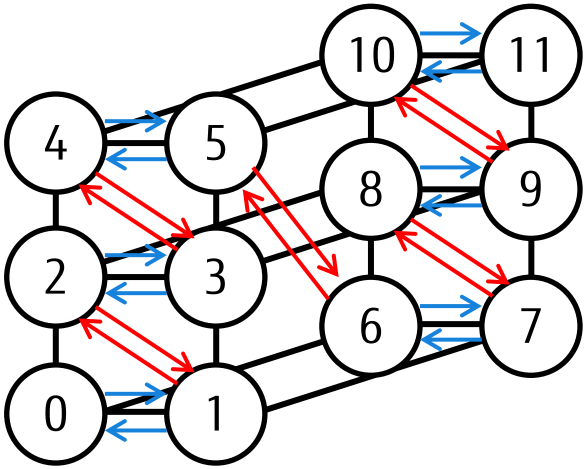
⇨ IMB exchange is a benchmark test that exchanges messages with adjacent rank.

One-dimensional torus (84 processes)



In a one-dimensional torus, there is only adjacent communication, so no contention occurs and communication performance does not degrade.

Three-dimensional torus (2x3x14 processes)



The red paths (such as between 1 and 2) use the paths for other communication, so performance degrades.

Trunking

FUJITSU

What is trunking?

- Trunking uses up to four Tofu network interfaces, and divides and transfers data by using multiple paths at the same time.

How to use trunking

- Specify --mca common_tofu_use_multi_path N.

Value	Meaning
1	Specifies communication using multiple communication paths (i.e., trunking) in point-to-point communication.
0	Specifies not to use multiple communication paths in point-to-point communication. The default value of this parameter is 0.

You can control the number of network interfaces used.

- Specify --mca common_tofu_max_tnis N.

Value	Meaning
1 or greater	Specifies the upper limit on the number of network interfaces used.
-1	Uses the maximum number of available network interfaces. The default value of this parameter is -1.

Eager and Rendezvous

- The Eager and Rendezvous communication methods are used for MPI point-to-point communication.
  - Eager communication method (suited for short messages)
    - Communication goes through send and receive buffers.
    - The message length used for communication is relatively short.
    - Asynchronous communication proceeds as long as the communication buffer has free space.
    - Copying of the send memory buffer and copying to the receive memory buffer occur.
  - Rendezvous communication method (suited for long messages)
    - Control communication to notify the other end of the send-receive location occurs internally.
    - If the first address of the send-receive data represents a continuous area, sending is done directly using RDMA.
  - The switching threshold is set to 45,352 Bytes (which varies depending on the number of communication hops).
  - You can change the switching threshold by using the MCA parameter btl_tofu_eager_limit.

Cost of Reception Wait (Profiler Results)

- Cost of reception wait as shown by the profiler
  - If ptlib_read_mrq and mca_btl_tofu_component_progress appear near the top, the reception wait event has occurred.

```
    Time(S)        Start     End
--------------- snip --------------
   324.5029          --       --   Application
--------------- ---- --------------
   116.8183          --       --   mca_btl_tofu_component_progress
    59.0181          --       --   ptlib_read_mrq
```

- Cause
  - There is a load imbalance.
    - The processing cost for only specific processes is high.

Acceleration of Collective Communication

- Tofu Barrier Communication
- Tofu-dedicated Algorithms for Collective Communication

Tofu Barrier Communication

FUJITSU

- The Tofu interconnect provides barrier communication at the hardware level.
  - Tofu hardware barrier resources are allocated to up to eight communicators.
    - The target communicators have a size of four or more.
  - The target MPI functions are as follows:
    - MPI_Barrier
    - MPI_[Reduce|Allreduce] + MPI_SUM + 1 element of floating-point type/complex type
    - MPI_[Reduce|Allreduce] + MPI_[SUM|MAX|MIN] + 1 element of integer type, etc.
    - MPI_Bcast + 1 element of basic datatypes other than complex type, etc.
  - This can also be used to generate multiple MPI processes in a node.
    - In a node, software is used. Between nodes, hardware is used.
    - To use a sub-communicator, it must meet all of the following conditions.
      - The parent of the sub-communicator is MPI_COMM_WORLD.
      - The sub-communicator was created by the MPI_Comm_split function.
      - The sub-communicator has color=0.
  - If resources become insufficient, no resources are allocated.
    - As a rough guide to resource consumption when one communicator is created, consumption is $2\log_2 N$ (where N is the number of nodes), and the upper limit is 56.

Problem in Tofu Barrier Communication + MPI_Bcast

FUJITSU

- Different type signatures may result in a deadlock. Essentially, this means the program is incorrect.
  - The MCA parameter coll_tbi_use_on_bcast has been prepared as a workaround.

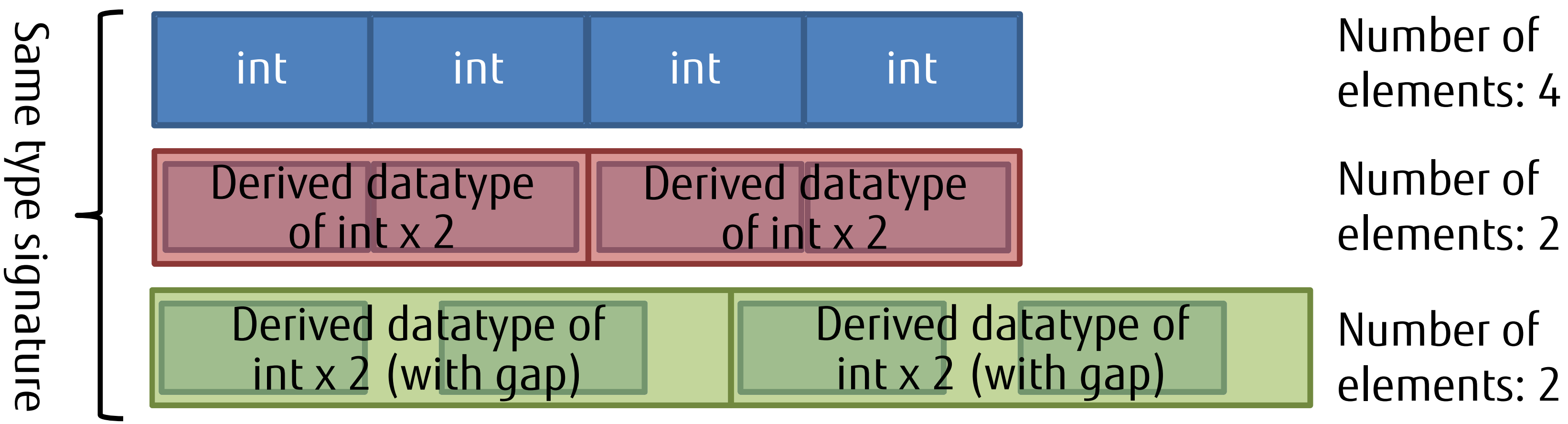
```
#include <mpi.h>

int main(int argc, char *argv[])
{
   long   sbuf = 0x0000000100000002L;
   int    rbuf[2];
   int    rank;
   MPI_Datatype   newtype;

   MPI_Init(&argc, &argv);

   MPI_Type_vector(1, 2, 2, MPI_INT, &newtype);
   MPI_Type_commit(&newtype);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank == 0){
      MPI_Bcast(&sbuf, 1, MPI_LONG, 0, MPI_COMM_WORLD);
   }else{
      MPI_Bcast(rbuf, 1, newtype, 0, MPI_COMM_WORLD);
   }

   MPI_Finalize();
}
```

The type signatures are different.

If --mca coll_tbi_use_on_bcast 0 is specified, rbuf = {1,2} is set.

If --mca coll_tbi_use_on_bcast is not specified, a deadlock occurs.

- MPI_Bcast and MPI_Ibcast allow you to use send and receive buffers with different data types and numbers of elements between ranks when their type signatures are the same. The MPI library uses a secure algorithm by default for operation even with different data types and numbers of elements.
  - The MCA parameter coll_tuned_bcast_same_count has been prepared so that a high-speed algorithm can be used when the number of elements is the same across all ranks.
  - If the user program can guarantee that the number of elements is the same, specify this MCA parameter.



Acceleration of the MPI_Bcast or MPI_Ibcast Function

- If each rank of the communicator that participates in MPI_Bcast or MPI_Ibcast has the same number of elements, the specification of 1 in the following MCA parameter may accelerate the MPI_Bcast function.

- Specified parameter

Parameter	Value	Meaning
coll_tuned_bcast_same_count	1	Specifies communication using the same number of elements between ranks by the MPI_Bcast or MPI_Ibcast function.
	0	Specifies communication using different numbers of elements between ranks by the MPI_Bcast or MPI_Ibcast function. The default value of this parameter is 0.

Tofu-dedicated Algorithms for Collective Communication

Definition of Tofu-dedicated algorithms

- The algorithms are implemented with RDMA communication.
- They are topology-aware algorithms.
  - Virtual three-dimensional: Bcast, Allreduce, Allgather(v), and Alltoall
    - Bcast is aware also for virtual two-dimensional and virtual one-dimensional.
    - Allreduce is aware also for virtual two-dimensional.
  - Tofu six-dimensional: Alltoall
  - Not topology-aware: Gather(v) and Alltoall(v)

Conditions of Tofu-dedicated algorithms

- The conditions of each algorithm are prerequisites. When all the conditions are met, you can call the Tofu-dedicated algorithm.
- Even when all the conditions are met, performance may not be optimized, depending on the number of processes, number of nodes, and message length. Consequently, the Tofu-dedicated algorithm may not be called.

- Tofu-dedicated algorithms are categorized as follows:
  - A) Algorithms that are selected automatically according to conditions such as message size and shape
    - i. Algorithm that can be called only for one process (or two processes) in a node
    - ii. Algorithm that can be called for two or more processes in a node
  - B) Algorithms that are always callable due to a specified MCA parameter
    - Only MPI_Alltoall, which is the six-dimensional algorithm described in table (4), falls into this category.

- The conditions for calling an algorithm are described in the following tables.
  - (1) and (2): Conditions that enable calling of algorithm i in A)
  - (3)-a and (3)-b: Conditions that enable calling of algorithm ii in A)
  - (4): Conditions for calling the algorithm in B)

- If the conditions for calling both algorithms i and ii are met, i is selected preferentially.

Tofu-dedicated Algorithm for Collective Communication (1)

FUJITSU

	MPI_Bcast	MPI_Allreduce MPI_Reduce	MPI_Allgather MPI_Allgatherv
Common conditions	■ The number of processes in a node is 1. ■ Jobs with a 3-dimensional shape are executed. ■ The communicator shape is a 3-dimensional rectangular parallelepiped and the process is arranged in all nodes. ■ The data type of the send and receive buffers is a basic datatypes. ■ The product of the following two values is a multiple of 4: - Size of the data type to send or receive - Number of elements (count/scount/rcount)		
Specific conditions	■ The size of each axis of the communicator is at least 2. (The shape of the communicator is 2 x 2 x 2 or larger.)	■ MPI_IN_PLACE is not specified. ■ Predefined operators (MAXLOC and MINLOC can be used only with the following predefined data types(Note 1).) Note 1 : Data types that can be used by Tofu-dedicated algorithms with MAXLOC/MINLOC - Fortran MPI_2INTEGER,MPI_2REAL,MPI_2DOUBLE_PRECISION - C MPI_FLOAT_INT, MPI_2INT - C++ MPI::TWOINT, MPI::FLOAT_INT, MPI::TWOINTEGER, MPI::TWOREAL, MPI::TWODOUBLE_PRECISION ■ The size of each axis of the communicator is at least 2. (The shape of the communicator is 2 x 2 x 2 or larger.)	■ The size of the send buffer is up to about 16 MiB * number of TNIs(Note 2) * communicator size. ■ Each element of the send and receive buffers is located on a 4-byte boundary (applicable only to MPI_Allgatherv). Note2 : Upper limit for TNI specified by MCA parameter comm_tofu_max_tnis.
Feature	■ Suited for medium-length/long messages	■ Suited for long messages	■ Suited for long messages

Tofu-dedicated Algorithm for Collective Communication (2)

FUJITSU

	MPI_Alltoall	MPI_Alltoall MPI_Alltoallv
Common conditions	■ The data type of the send and receive buffers is a basic datatypes. ■ The product of the following two values is a multiple of 4: - Size of the data type to send or receive - Number of elements (scount/rcount)	
Specific conditions	■ The number of processes in a node is 1. ■ Jobs with a 3-dimensional shape are executed. ■ The communicator shape is a 3-dimensional rectangular parallelepiped and the process is arranged in all nodes. ■ The length on each axis of a 3-dimensional shape is an even number. ■ The number of processes is the same as MPI_COMM_WORLD. ■ The receive buffer size is up to 32 MiB * communicator size.	■ The number of processes in a node is 1 or 2. ■ Each element of the send and receive buffers is located on a 4-byte boundary (applicable only to MPI_Alltoallv).
Feature	■ Suited for long messages	■ Suited for short to medium-length messages

Tofu-dedicated Algorithm for Collective Communication (3)-a

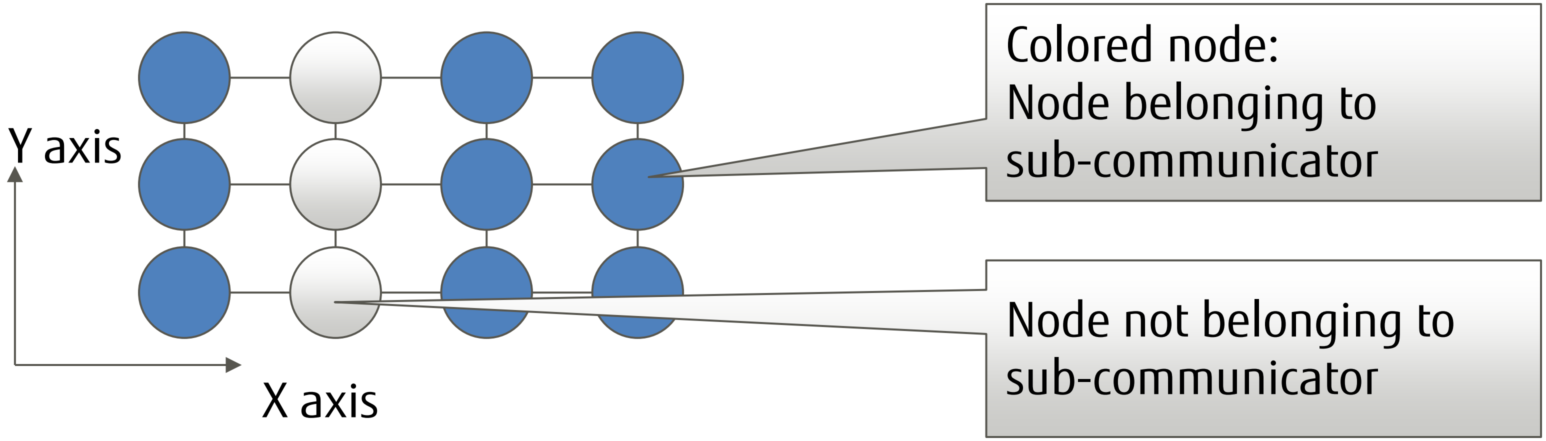
FUJITSU

	MPI_Bcast	MPI_Allreduce	MPI_Allgather	MPI_Allgatherv
Common conditions	■ The data type of the send and receive buffers is a basic datatypes. ■ The product of the following two values is a multiple of 4: - Size of the data type to send or receive - Number of elements (count/scount/rcount)			
Specific conditions	■ The communicator shape is a 3-dimensional rectangular parallelepiped, 2-dimensional rectangle, or 1-dimensional shape. ■ If the shape is a rectangular parallelepiped, the size of each axis of the 2 or more axes of the communicator is at least 2.	■ Jobs with a 3- or 2-dimensional shape are executed. ■ The communicator shape is a 3- or 2-dimensional rectangle. • The size of each axis of the 2 or more axes of the communicator is at least 2. ■ Predefined operators (MAXLOC and MINLOC can be used only with the following predefined data types(*).) * Data types that can be used by Tofu-dedicated algorithms with MAXLOC/MINLOC - Fortran MPI_2INTEGER,MPI_2REAL, MPI_2DOUBLE_PRECISION - C MPI_FLOAT_INT, MPI_2INT - C++ MPI::TWOINT, MPI::FLOAT_INT, MPI::TWOINTEGER, MPI::TWOREAL, MPI::TWODOUBLE_PRECISION	■ Jobs with a 3-dimensional shape are executed. ■ The size of each axis of the communicator is at least 2. (The shape of the communicator is 2 x 2 x 2 or larger.)	■ Jobs with a 3-dimensional shape are executed. ■ The size of each axis of the communicator is at least 2. (The shape of the communicator is 2 x 2 x 2 or larger.) ■ Each element of the send and receive buffers is located on a 4-byte boundary.
Feature	■ Suited for medium-length/long messages	■ Suited for medium-length messages	■ Suited for medium-length/long messages	■ Suited for medium-length/long messages

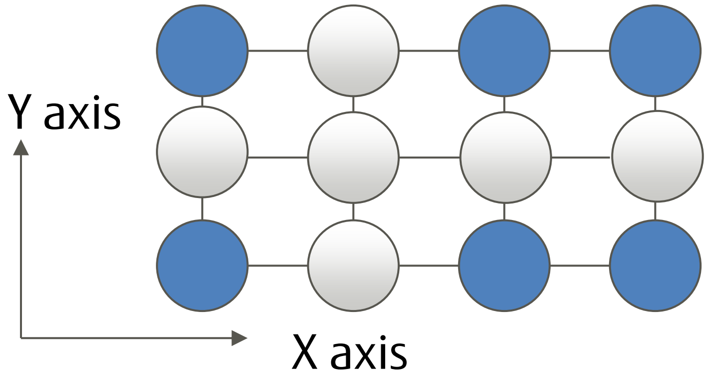
For allreduce in 2 dimensions, a different advantageous algorithm may be selected for execution with a small number of nodes.

- Condition for determining whether the communicator is a rectangular parallelepiped, a rectangle, or another shape
  - If the virtual X, Y, or Z axis meets the following condition, the communicator is a rectangular parallelepiped.
    - On a virtual axis (e.g., X=0,1,2…N)
      - All the coordinates in the same way have nodes that belong to the communicator.

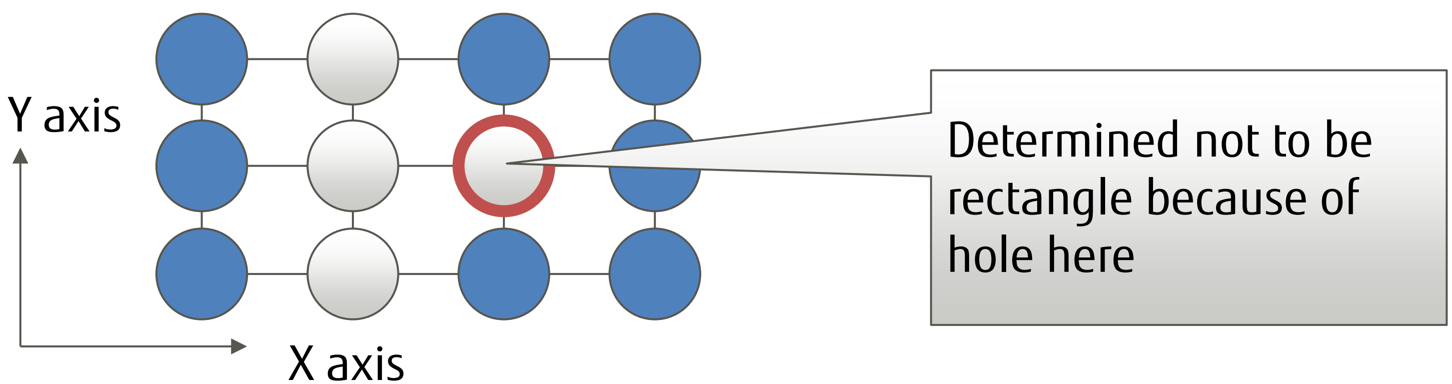
- The following example shows a job executed at 4x3 or 4x3x1.
  - Example with a sub-communicator determined to be a two-dimensional rectangle (1)
    - At X=1, none of the nodes belongs to the communicator.
    - At X=0, 2, and 3, all the nodes belong to the communicator.



- The following examples show jobs executed at 4x3 or 4x3x1.
  - Example with a sub-communicator determined to be a rectangle (2)
    - As viewed from the X axis, only Y=0 and Y=2 belong to the communicator.
    - As viewed from the Y axis, only X=0, 2, and 3 belong to the communicator.



  - Example with a sub-communicator determined not to be a two-dimensional rectangle



Tofu-dedicated Algorithm for Collective Communication (3)-b

	MPI_Alltoall	MPI_Alltoallv	MPI_Gather	MPI_Gatherv
Common conditions	■ The data type of the send and receive buffers is a basic datatypes. ■ The product of the following two values is a multiple of 4: - Size of the data type to send or receive - Number of elements (count/scount/rcount)			
Specific conditions	■ None	■ Each element of the send and receive buffers is located on a 4-byte boundary.	■ None	■ Each element of the send and receive buffers is located on a 4-byte boundary.
Feature	■ Suited for medium-length/long messages	■ Suited for all messages	■ Suited for all messages	■ Suited for medium-length/long messages

Tofu-dedicated Algorithm for Collective Communication (4)

	MPI_Alltoall (6-dimensional algorithm)
Conditions	■ The number of processes in a node is 1. ■ The communicator shape is a 6-dimensional rectangular parallelepiped. ■ The data type of the send and receive buffers is a basic datatypes. ■ The product of the following two values is a multiple of 4: - Size of the data type to send or receive - Number of elements (scount/rcount) ■ The MCA parameter coll_tuned_use_6d_algorithm is specified as 1.
Feature	■ Algorithm that is callable due to a specified MCA parameter ■ Suited for medium-length/long messages ■ Advantageous to performance in the following cases: ■ The message length is 8 to 10 KiB or longer. ■ The job size is at the level of thousands of nodes or greater. ■ The MCA parameter coll_tuned_prealloc_size is specified.

MPI Statistical Information (MCA Parameter)

How to use the parameter

- Specify a value for the MCA parameter mpi_print_stats.

Value	Meaning
0	Does not output MPI statistical information. The default value of this parameter is 0.
1	Outputs the tabulated results of MPI statistical information on all parallel processes to the standard error output. The results are output by the process of rank 0 in MPI_COMM_WORLD.
2	Outputs MPI statistical information of each parallel process to the standard error output. The results are output by each parallel process. If you want the output for a specific parallel process, you can specify it with the MCA parameter mpi_print_stats_ranks.
3	Outputs the tabulated results of MPI statistical information section specifying on all parallel processes to the standard error output. The results are output by the process of rank 0 in MPI_COMM_WORLD.
4	Outputs MPI statistical information of each parallel process to the standard error output. The results are output by each parallel process. If you want the output for a specific parallel process, you can specify it with the MCA parameter mpi_print_stats_ranks.

MPI Statistical Information Section Specifying

FUJITSU

- MPI statistical information section specifying
  - Measure statistical data from a user-specified location.
  - Specify 3 or 4 for the MCA parameter mpi_print_stats.

Function name	Function
FJMPI_Collection_start	Starts MPI statistical information measurement
FJMPI_Collection_stop	Stops MPI statistical information measurement
FJMPI_Collection_print	Prints MPI statistical information measurement
FJMPI_Collection_clear	Initializes MPI statistical information

MPI Statistical Information Results

FUJITSU

Example of output results

```
===================================================================
/***************** MPI Statistical Information *****************/
===================================================================

------------------------- MPI Information -------------------------
Dimension                    3
Shape                      2x3x2

---------------------- MPI Memory Usage (MiB) ---------------------
                            MAX               MIN              AVE
Estimated_Memory_Size     49.81 [    1]     49.81 [    0]     49.81
```

Communication information on point-to-point communication

```
------------------- Per-peer Communication Count -------------------
                            MAX               MIN              AVE
In_Node                       0 [    0]         0 [    0]       0.0
Neighbor                  21140 [    0]      5285 [    3]    9694.8
Not_Neighbor              10587 [    3]      5285 [    0]    7937.4
Total_Count               26442 [    1]     15872 [    2]   17632.2
Connection                   11 [    0]        11 [    0]      11.0
Max_Hop                       3 [    0]         3 [    0]       3.0
Average_Hop                1.82 [    0]      1.82 [    0]      1.82

----------------- Per-peer Transmission Size (MiB) -----------------
                            MAX               MIN              AVE
In_Node                    0.00 [    0]      0.00 [    0]      0.00
Neighbor                 335.91 [    0]     83.98 [    3]    153.96
Not_Neighbor             167.96 [    3]     83.98 [    0]    125.97
Total_Size               419.89 [    1]    251.94 [    2]    279.93

----------------- Per-protocol Communication Count -----------------
                            MAX               MIN              AVE
Eager                     20037 [    1]     12029 [    2]   13362.2
Rendezvous                 6405 [    0]      3843 [    2]    4270.0
Hasty_Rendezvous              0 [    0]         0 [    0]       0.0
Unexpected_Message            8 [    1]         2 [    0]       2.6

------------------- Barrier Communication Count -------------------
                            MAX               MIN              AVE
Tofu                          8 [    0]     51218 [    0]   51218.0
Soft                          6 [    0]        16 [    2]      22.7
```

Barrier (software or hardware)

```
----------- Tofu Barrier Collective Communication Count -----------
                            MAX               MIN              AVE
Bcast                           [    0]      2046 [    0]    2046.0
Reduce                          [    0]      2003 [    0]    2003.0
Allreduce                       [    0]      2046 [    0]    2046.0
```

Reduction (software or hardware)

Collective communication (Tofu-dedicated or non-Tofu-dedicated)

```
--------- 6D-Tofu-specific Collective Communication Count ---------
                            MAX               MIN              AVE
Alltoall                      0 [    0]         0 [    0]       0.0

----------- Tofu-specific Collective Communication Count ----------
                            MAX               MIN              AVE
Bcast                      1281 [    0]      1281 [    0]    1281.0
Reduce                     1281 [    0]      1281 [    0]    1281.0
Gather                     5334 [    0]      5334 [    0]    5334.0
Allreduce                  1281 [    0]      1281 [    0]    1281.0
Alltoall                   5285 [    0]      5285 [    0]    5285.0
Alltoallv                  5285 [    0]      5285 [    0]    5285.0
Allgather                  3282 [    0]      3282 [    0]    3282.0
Allgatherv                 5285 [    0]      5285 [    0]    5285.0

--------- Non-Tofu-specific Collective Communication Count --------
                            MAX               MIN              AVE
Bcast                      2010 [    0]      2004 [    2]    2005.0
Reduce                     2001 [    0]      2001 [    0]    2001.0
Gather                        0 [    0]         0 [    0]       0.0
Allreduce                  1927 [    0]      1909 [    2]    1912.0
Alltoall                      0 [    0]         0 [    0]       0.0
Alltoallv                     0 [    0]         0 [    0]       0.0
Allgather                  2003 [    0]      2003 [    0]    2003.0
Allgatherv                    0 [    0]         0 [    0]       0.0
```

```
----- Per-protocol Nonblocking/Persistent Communication Count -----
MAX MIN AVE
Eager                         0 [ 0]            0 [ 0]           0.0
Rendezvous                    0 [ 0]            0 [ 0]           0.0
Hasty_Rendezvous              0 [ 0]            0 [ 0]           0.0
Collective                    0 [ 0]            0 [ 0]           0.0

-- Per-protocol Nonblocking/Persistent Communication Count Started in
Wait --
MAX MIN AVE
Eager               0 [ 0] 0 [ 0] 0.0
Rendezvous          0 [ 0] 0 [ 0] 0.0
Hasty_Rendezvous    0 [ 0] 0 [ 0] 0.0
```

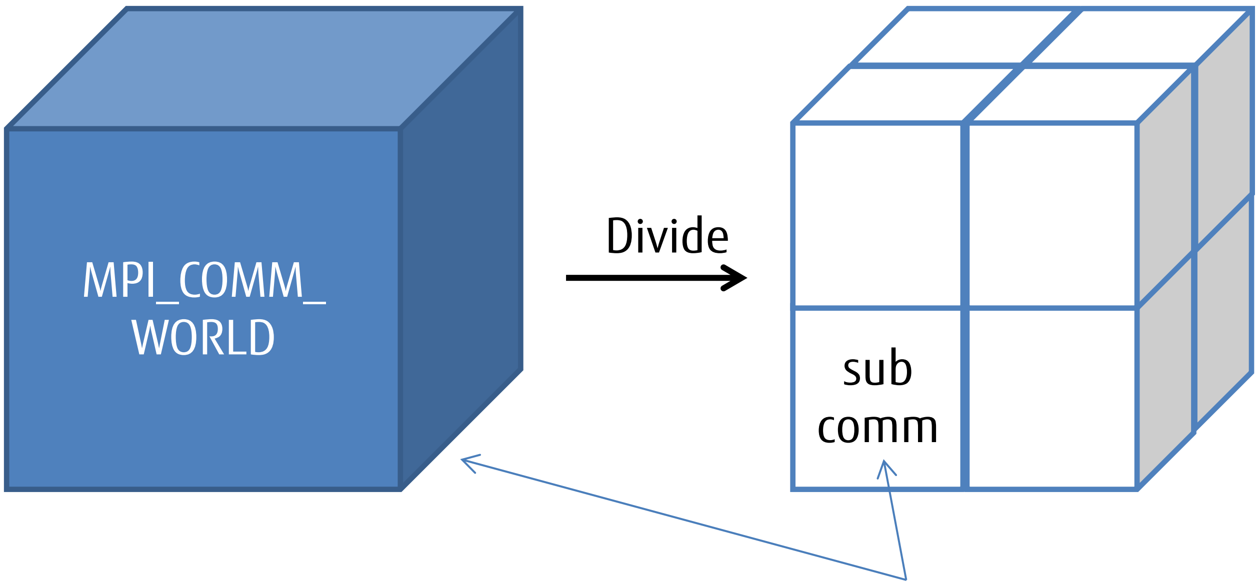
Nonblocking communication

```
------------------------- Process Mapping -------------------------
(0,0,0)                   0
(1,0,0)                   1
(0,1,0)                   2
(1,1,0)                   3
```

Possibility of Improvement Based on Statistics

FUJITSU

- Whether a Tofu-specific algorithms can be called
  - Low numbers in Tofu-specific Collective Communication Count
    - Is the job executed in three dimensions? (Excluding Alltoall(v) and Gather(v), which are for relatively short messages)
    - Is the communicator a three-dimensional rectangular parallelepiped or two-dimensional rectangle? (Excluding Alltoall(v) and Gather(v))



An applied condition for most Tofu-dedicated algorithms is that the communicator must be a 3-dimensional rectangular parallelepiped.

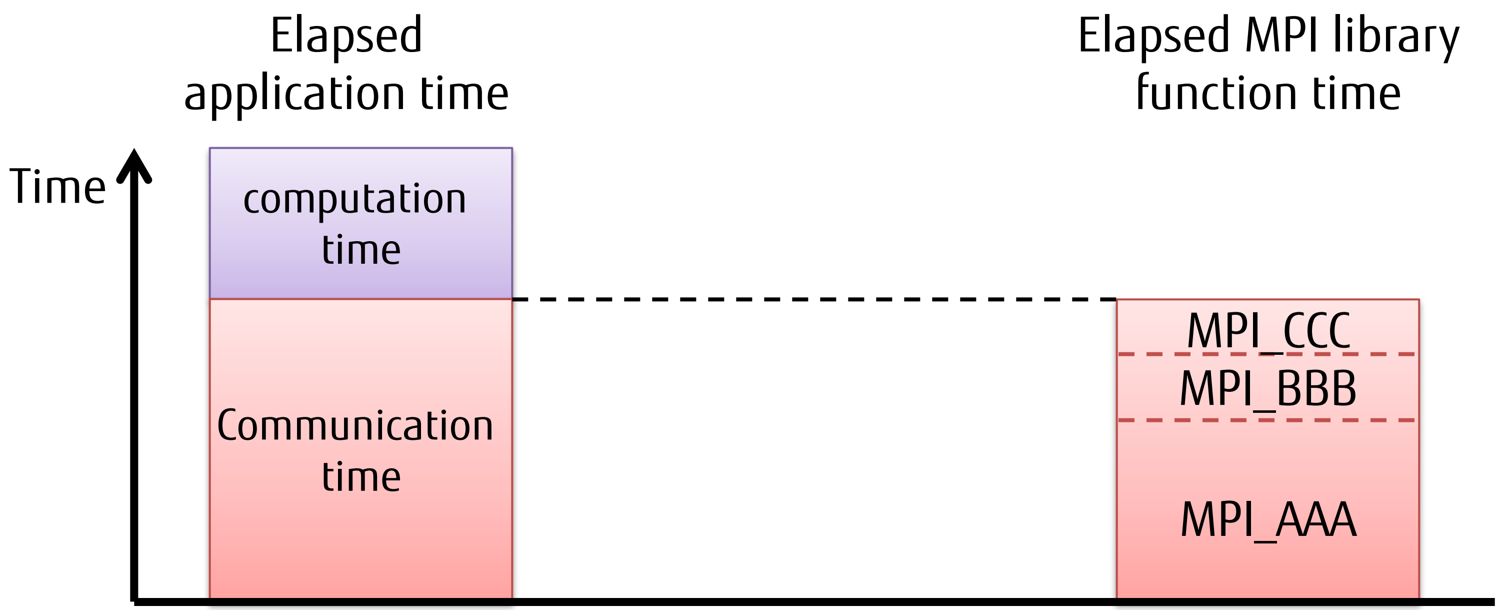
Tuning Examples

- Problem detection guidelines for communication times
- Overview of tuning examples
  - Effective nonblocking communication using four TNIs
  - Use of trunking
  - Examples of overlapping computations and communications
    - Facilitating communication by inserting MPI_Testall
    - Implementing a communication-dedicated thread using OpenMP
  - Improvement through the data types used
  - Communication Using the Basic/Derived Datatype
  - Use of assistant cores
  - Example of differences in performance with a specified shape

  * See the "MPI User's Guide 4.2 MCA Parameters" for details.

Problem Detection Guidelines for Communication Times (1/2)

- Step 1: Check ratio of computation time to communication time
- Step 2: Identify high-cost MPI functions



You can obtain a breakdown of the application execution time by using the fipp command.

If the ratio of communication time to elapsed application time is high, there may be a communication problem.

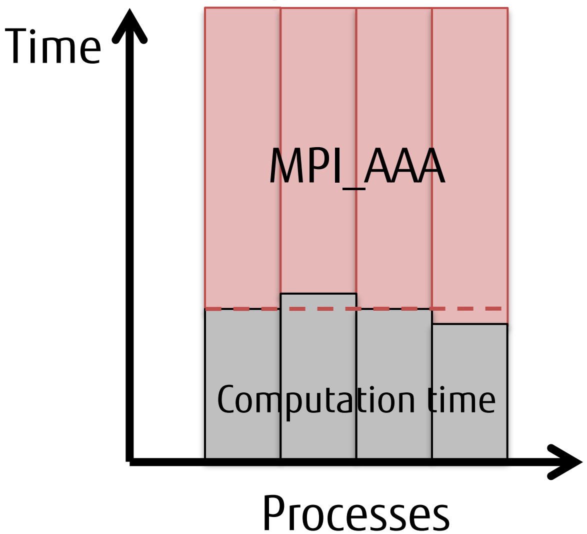
Identify high-cost MPI functions.
Example: Function type (point-to-point communication, collective communication), data type (basic datatypes, derived datatype)

Problem Detection Guidelines for Communication Times (2/2)

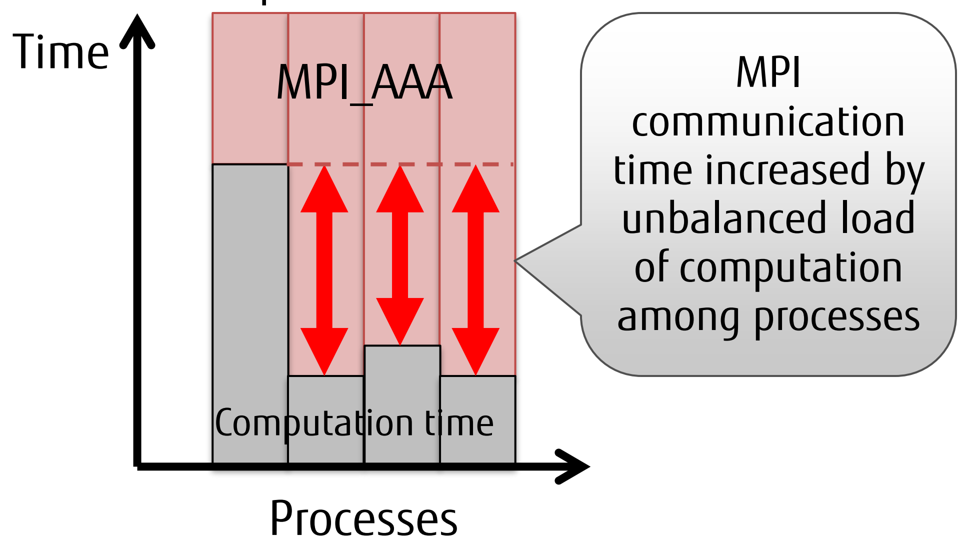
FUJITSU

- Step 3: Check computation time of each process

MPI communication time and computation time of each process



MPI communication time and computation time of each process



You can obtain the computation time of each process by using the fipp command.

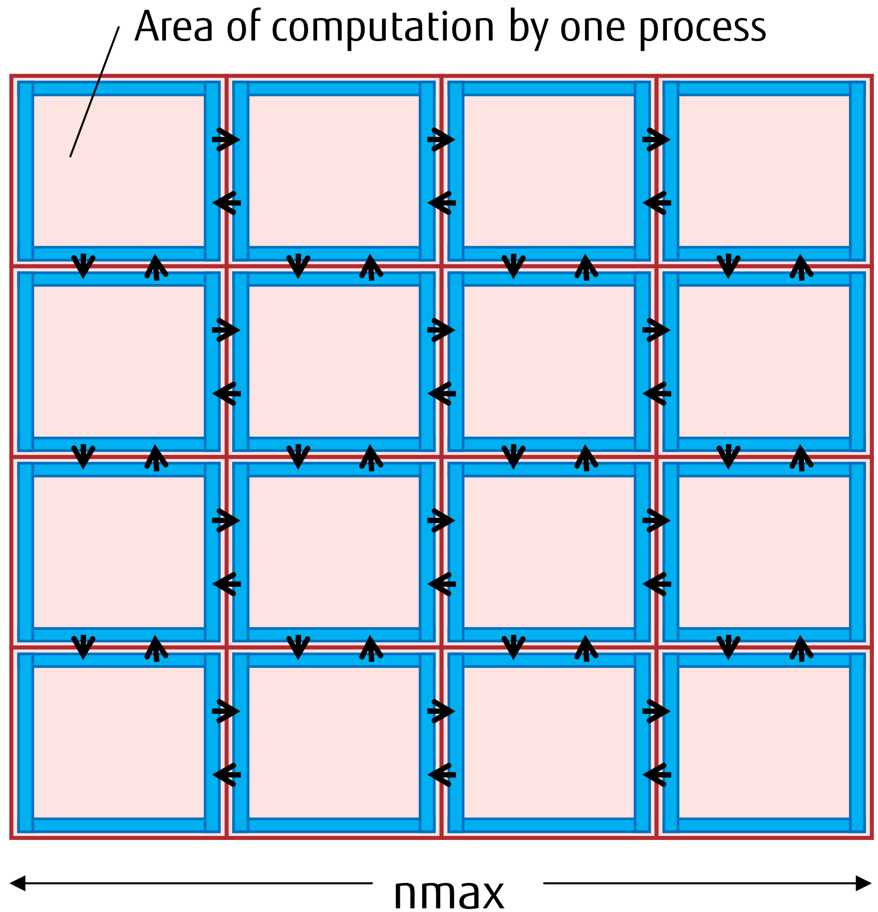
If computation time is almost balanced among processes, optimize.	If computation time is unbalanced among processes, optimize computation to balance loads.

Overview of Tuning Examples

FUJITSU

- This section describes tuning examples and the processing of parallel application that are tuning targets.
  - Assume an application solves two-dimensional differential equations using the Jacobi method.
  - The assumed sleeve area communication is a two-dimensional space (nmax x nmax).
  - One process sends and receives data with the processes on the left, right, top, and bottom. There are non-periodic boundary conditions.
  - The code implemented with nonblocking communication is a tuning target. (See the next page.)

Example: Division of spaces and sleeve communication for using 16 processes



Implementation with Nonblocking Communication

FUJITSU

```
do  iter = 1, 10
 do j = 1, ny
  bxs(j,1) = a(1,j)
  bxs(j,2) = a(nx,j)
 enddo

 do i = 1, nx
  bys(i,1) = a(i,1)
  bys(I,2) = a(i,ny)
 enddo

 call mpi_irecv( bxr(1,2), ny, mpi_real8, left, 1, comm2d, req(1), err )
 call mpi_irecv( bxr(1,1), ny, mpi_real8, right, 2, comm2d, req(2), err )

 call mpi_irecv( byr(1,2), nx, mpi_real8, down, 1, comm2d, req(3), err )
 call mpi_irecv( byr(1,1), nx, mpi_real8, up,  2, comm2d, req(4), err )

 call mpi_isend( bxs(1,2), ny, mpi_real8, right, 1, comm2d, req(5), err )
 call mpi_isend( bxs(1,1), ny, mpi_real8, left, 2, comm2d, req(6), err )

 call mpi_isend( bys(1,2), nx, mpi_real8, up, 1, comm2d, req(7), err )
 call mpi_isend( bys(1,1), nx, mpi_real8, down, 2, comm2d, req(8),err )

 call mpi_waitall( 8, req, mpi_statuses_ignore, err )
 if( left .ne. mpi_proc_null ) then
  do j = 1, ny
    a(0,j)    = bxr(j,2)
  enddo
 endif

 if( right .ne. mpi_proc_null ) then
  do j = 1, ny
   a(nx+1,j) = bxr(j,1)
  enddo
 endif
```

Communication

```
 if( up .ne. mpi_proc_null ) then
  do i = 1, nx
    a(i,ny+1) = byr(i,1)
   enddo
 endif

 if( down .ne. mpi_proc_null ) then
  do i = 1, nx
    a(i,0)    = byr(i,2)
   enddo
 endif

 do j = 1, ny
  do i = 1, nx
   b(i,j) = 0.25d0 * ( a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j) )
  enddo
 enddo

 norm(1) = 0.0d0
 norm(2) = 0.0d0
 do j = 1, ny
  do i = 1, nx
   a(i,j) = b(i,j)
   norm(1) = norm(1) + ( a(I,j) – b(I,j) ) ** 2
   norm(2) = norm(2) + b(I,j) ** 2
  enddo
 enddo

enddo
```

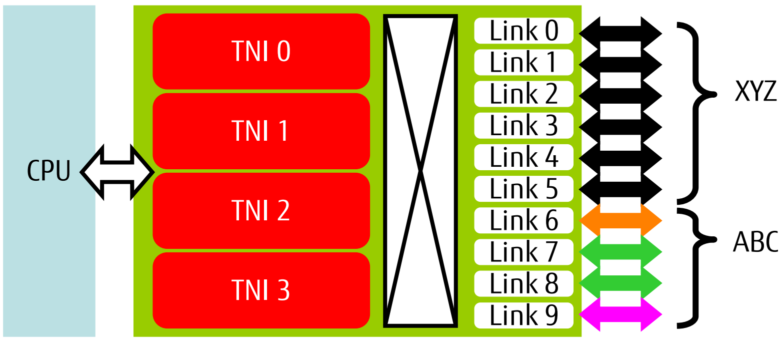
Computation

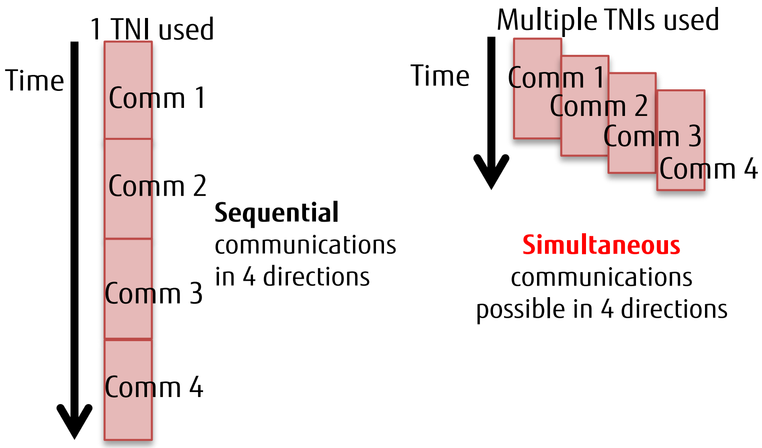
Effective Nonblocking Communication Using Four TNIs

FUJITSU

- Multiple TNIs (four TNIs) available for Tofu communication

  Four TNIs are mounted, enabling four sends and four receives at the same time.





Simultaneous nonblocking communications in 4 directions
⇨ Communication overlap (simultaneous communications using 4 TNIs) resulting in
significantly improved performance

- Simultaneous communications in multi directions by merging mpi_waitall functions into one

Communication for each 1 direction

```
! Adjacent communication in K direction
  call mpi_irecv(p(1,1,kmax),1,ijvec, npz(2), 1,mpi_comm_cart,ireq(3),ierr)
  call mpi_irecv(p(1,1,1),1,ijvec, npz(1),2,mpi_comm_cart,ireq(2),ierr)
  call mpi_isend(p(1,1,2),1,ijvec,npz(1),1,mpi_comm_cart,ireq(0),ierr)
  call mpi_isend(p(1,1,kmax-1),1,ijvec,npz(2),2,mpi_comm_cart,ireq(1), ierr)
  call mpi_waitall(4,ireq,ist,ierr)

! Adjacent communication in J direction
  call mpi_irecv(p(1,1,1),1,ikvec,npy(1),2,mpi_comm_cart,ireq(3),ierr)
  call mpi_irecv(p(1,jmax,1),1,ikvec,npy(2),1,mpi_comm_cart,ireq(2),ierr)
  call mpi_isend(p(1,2,1),1,ikvec,npy(1),1,mpi_comm_cart,ireq(0),ierr)
  call mpi_isend(p(1,jmax-1,1),1,ikvec,npy(2),2,mpi_comm_cart,ireq(1),ierr)
  call mpi_waitall(4,ireq,ist,ierr)
```

Simultaneous communications in 2 directions

```
! Adjacent communication in K direction
  call mpi_irecv(p(1,1,kmax),1,ijvec, npz(2), 1,mpi_comm_cart,ireq(3),ierr)
  call mpi_irecv(p(1,1,1),1,ijvec, npz(1),2,mpi_comm_cart,ireq(2),ierr)
  call mpi_isend(p(1,1,2),1,ijvec,npz(1),1,mpi_comm_cart,ireq(0),ierr)
  call mpi_isend(p(1,1,kmax-1),1,ijvec,npz(2),2,mpi_comm_cart,ireq(1), ierr)

! Adjacent communication in J direction
  call mpi_irecv(p(1,1,1),1,ikvec,npy(1),2,mpi_comm_cart,ireq(7),ierr)
  call mpi_irecv(p(1,jmax,1),1,ikvec,npy(2),1,mpi_comm_cart,ireq(6),ierr)
  call mpi_isend(p(1,2,1),1,ikvec,npy(1),1,mpi_comm_cart,ireq(4),ierr)
  call mpi_isend(p(1,jmax-1,1),1,ikvec,npy(2),2,mpi_comm_cart,ireq(5),ierr)
!
  call mpi_waitall(8,ireq,ist,ierr)
```

* Communications using multiple TNIs improve performance when the communication method is Rendezvous.

Use of Trunking

FUJITSU

- Example of throughput improvement through trunking
  - Sometimes, you can expect communications using multiple communication paths, that is, trunking, to improve throughput in communication performance.
  - To use trunking, specify 1 for the MCA parameter common_tofu_use_multi_path.
  - The performance of the PingPong benchmark test may double by using the trunking. (Execution with two nodes and two processes)

  * Depending on the application program or other communication environment conditions, the effects of trunking may not be obtained and communication contention may occur, resulting in performance degradation. Take sufficient care about using trunking.

■ PingPong source code example

```
if (rank == 0) {
  for(i=0; i<ITERATIONS; i++) {
    MPI_Send (...);
    MPI_Recv  (...);
  }
}
else if (rank == 1) {
  for(i=0; i<ITERATIONS; i++) {
    MPI_Recv  (...);
    MPI_Send (...);
  }
}
```

■ Results when multiple communication paths are not used (common_tofu_use_multi_path 0)

```
#-----------------------------------------------
# Benchmarking PingPong
# # processes = 2
#-----------------------------------------------
   #bytes  #repetitions      t[usec]     Mbytes/sec
 4194304             10       429.95        9303.41
```

■ Results when communication uses multiple communication paths (common_tofu_use_multi_path 1)

```
#-----------------------------------------------
# Benchmarking PingPong
# # processes = 2
#-----------------------------------------------
   #bytes  #repetitions      t[usec]     Mbytes/sec
 4194304             10       211.17       18941.69
```

Prerequisites to Overlapping Computations and Communications

- Blocking communication: Returning from the send/receive function after the communication is completed
  - The computation and communication can never overlap.

```
call mpi_sendrecv(bxsr,       ny,  mpi_real8,  right,  1,
                  bxrr,       ny,  mpi_real8,  left,   1,  comm2d,  mpi_status_ignore, err )
call mpi_sendrecv(bxsl,       ny,  mpi_real8,  left,   2,
                  bxrl,       ny,  mpi_real8,  right,  2,  comm2d,  mpi_status_ignore, err )
call mpi_sendrecv(bysu,       nx,  mpi_real8,  up,     1,
                  byru,       nx,  mpi_real8,  down,   1,  comm2d,  mpi_status_ignore, err )
call mpi_sendrecv(bysd,       nx,  mpi_real8,  down,   2,
                  byrd,       nx,  mpi_real8,  up,     2,  comm2d,  mpi_status_ignore, err )
```

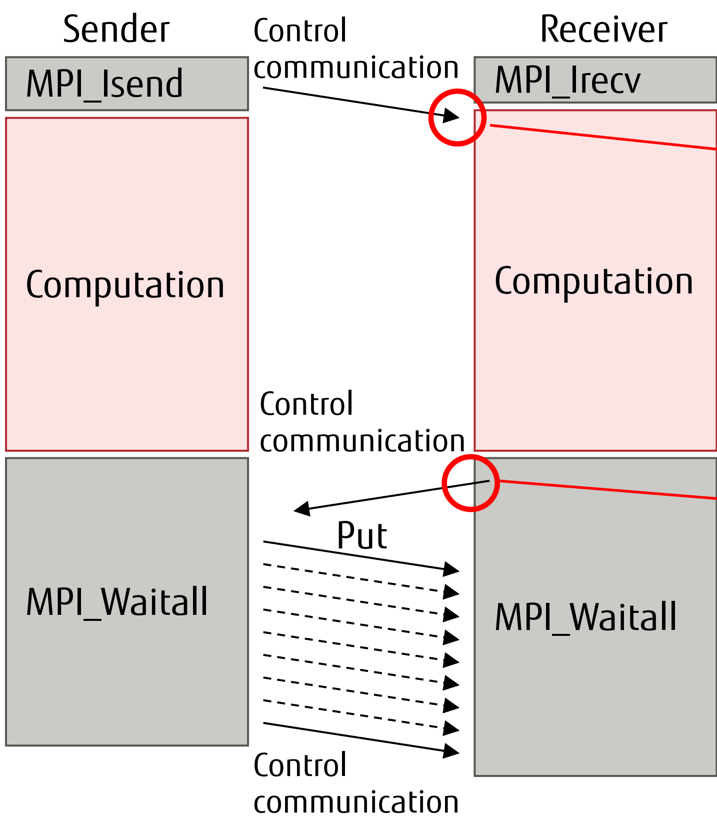
- Nonblocking communication: The send/recieve functions return after the communication begins
  - The computation and communication can overlap.

```
call mpi_irecv(bxrr,     ny,  mpi_real8,  left,   1,  comm2d, reqs(1), err )
call mpi_irecv(bxrl,     ny,  mpi_real8,  right,  2,  comm2d, reqs(2), err )
call mpi_irecv(byru,     nx,  mpi_real8,  down,   1,  comm2d, reqs(3), err )
call mpi_irecv(byrd,     nx,  mpi_real8,  up,     2,  comm2d, reqs(4), err )
call mpi_isend(bxsr,     ny,  mpi_real8,  right,  1,  comm2d, reqs(5), err )
call mpi_isend(bxsl,     ny,  mpi_real8,  left,   2,  comm2d, reqs(6), err )
call mpi_isend(bysu,     nx,  mpi_real8,  up,     1,  comm2d, reqs(7), err )
call mpi_isend(bysd,     nx,  mpi_real8,  down,   2,  comm2d, reqs(8), err )
call mpi_waitall( 8, reqs, mpi_statuses_ignore, err )
```

To overlap computations and communications, use nonblocking communication.

Computations and Communications not Overlapping

- Using the Rendezvous communication method with nonblocking communication, the receiver is already under computation when control communication arrives.



The next control communication cannot start when control communication arrives from MPI_Isend.

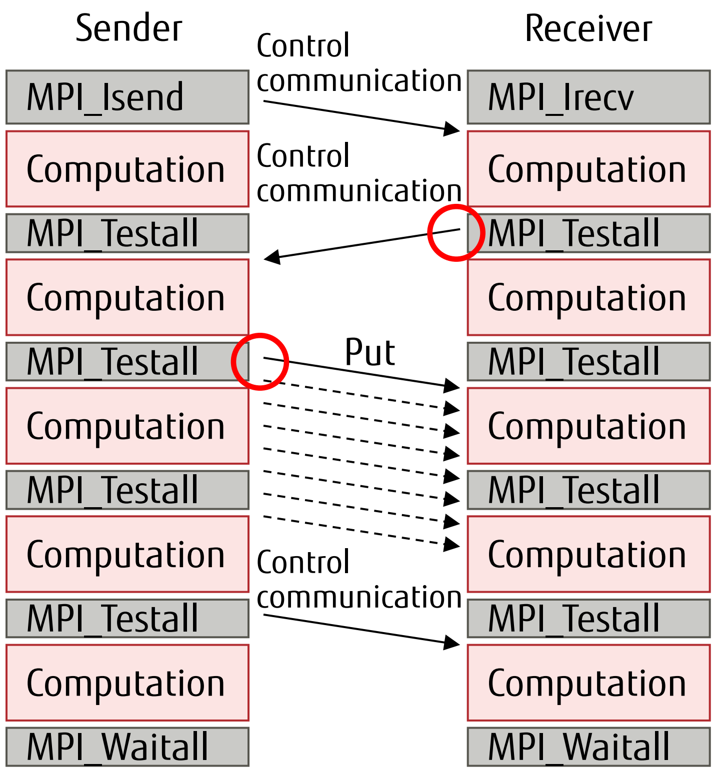
It is after MPI_Waitall function is called that the next control communication can start.

Communications and computations do not overlap.

* Basically, overlap is possible with the Eager communication method. However, if the sender calls Isend frequently within a short period and processing by the receiver is late, overlap may be impossible because of a receive buffer shortage.
* This example uses two processes, but the same applies even to point-to-point communication with multiple processes.
* Determine the communication method (Eager and Rendezvous) based on message length or MPI statistical information. For the relationship between message length and communication method, see the btl_tofu_eager_limit item in the *MPI User's Guide*.

Facilitating Communication by Inserting MPI_Testall

- An MPI_Testall function call during computation transfers control temporarily to the MPI library.
- Upon detecting the arrival and completion of communication, the MPI library gives an instruction for the next communication to the Tofu interconnect when calling the MPI_Testall function.



The MPI library gives an instruction for the next communication to the Tofu interconnect when calling the MPI_Testall function.
After the instruction, communication runs in the background of the computation.

* If MPI_Test is used instead of MPI_Testall, the number of control communications processed with one call is one.
* This example uses two processes, but the same applies even to point-to-point communication with multiple processes.

Computations and Communications not Overlapping

```
do  iter = 1, 10
 do j = 1, ny
  bxs(j,1) = a(1,j)
  bxs(j,2) = a(nx,j)
 enddo

 do i = 1, nx
  bys(i,1) = a(i,1)
  bys(I,2) = a(i,ny)
 enddo

call mpi_irecv( bxr(1,2), ny, mpi_real8, left, 1, comm2d, req(1), err )
call mpi_irecv( bxr(1,1), ny, mpi_real8, right, 2, comm2d, req(2), err )

call mpi_irecv( byr(1,2), nx, mpi_real8, down, 1, comm2d, req(3), err )
call mpi_irecv( byr(1,1), nx, mpi_real8, up,  2, comm2d, req(4), err )

call mpi_isend( bxs(1,2), ny, mpi_real8, right, 1, comm2d, req(5), err )
call mpi_isend( bxs(1,1), ny, mpi_real8, left, 2, comm2d, req(6), err )

call mpi_isend( bys(1,2), nx, mpi_real8, up, 1, comm2d, req(7), err )
call mpi_isend( bys(1,1), nx, mpi_real8, down, 2, comm2d, req(8),err )

call mpi_waitall( 8, req, mpi_statuses_ignore, err )
```

Communication

```
if( left .ne. mpi_proc_null ) then
    do j = 1, ny
      a(0,j)    = bxr(j,2)
    enddo
endif

if( right .ne. mpi_proc_null ) then
    do j = 1, ny
      a(nx+1,j) = bxr(j,1)
    enddo
endif
```

```
 if( up .ne. mpi_proc_null ) then
   do i = 1, nx
     a(i,ny+1) = byr(i,1)
   enddo
 endif

 if( down .ne. mpi_proc_null ) then
   do i = 1, nx
     a(i,0)    = byr(i,2)
   enddo
 endif

 do j = 1, ny
   do i = 1, nx
     b(i,j) = 0.25d0 * ( a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j) )
   enddo
 enddo

 norm(1) = 0.0d0
 norm(2) = 0.0d0
 do j = 1, ny
   do i = 1, nx
     a(i,j) = b(i,j)
     norm(1) = norm(1) + ( a(I,j) – b(I,j) ) ** 2
     norm(2) = norm(2) + b(I,j) ** 2
   enddo
 enddo
enddo
```

Computation

Facilitating Communication by Inserting MPI_Testall

```
do iter = 1, 10
 do j = 1, ny
  bxs(j,1) = a(1,j)
  bxs(j,2) = a(nx,j)
 enddo
 do i = 1, nx
  bys(I,1) = a(i,1)
  bys(I,2) = a(i,ny)
 enddo
 call mpi_irecv( bxr(1,2), ny, mpi_real8, left, 1, comm2d, req(1), err )
 call mpi_irecv( bxr(1,1), ny, mpi_real8, right, 2, comm2d, req(2), err )
 call mpi_irecv( byr(1,2), nx, mpi_real8, down, 1, comm2d,req(3),err )
 call mpi_irecv( byr(1,1), nx, mpi_real8, up, 2, comm2d, req(4), err )
 call mpi_isend( bxs(1,2), ny, mpi_real8, right, 1, comm2d,req(5),err )
 call mpi_isend( bxs(1,1), ny, mpi_real8, left,  2, comm2d, req(6), err )
 call mpi_isend( bys(1,2), nx, mpi_real8, up,  1, comm2d, req(7), err )
 call mpi_isend( bys(1,1), nx, mpi_real8, down, 2,comm2d,req(8),err )

 do j = 2, ny-1
  do i = 2, nx-1
   b(i,j) = 0.25d0 * ( a(i-1,j) + a(i,j-1) + a(i,j+1) + a(i+1,j) )
  enddo
 enddo
 norm(1) = 0.0d0
 norm(2) = 0.0d0
 do j = 2, ny-1
  do i = 2, nx-1
   norm(1) = norm(1) + ( b(i,j) - a(i,j) ) ** 2
   norm(2) = norm(2) + b(i,j) ** 2
  enddo
  if( done == 0 ) call mpi_testall( 8,req,done,mpi_statuses_ignore,err )
 enddo
 if( done .eq. 0 ) then
  call mpi_waitall( 8, req, mpi_statuses_ignore, err )
 endif
```

Computes first the part that is independent of the communication. Computations and communications overlap.

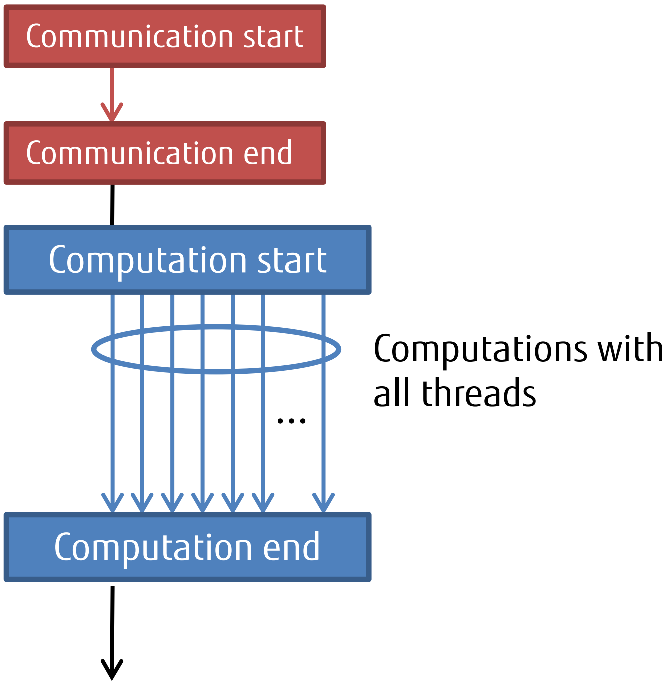
```
 if( left .ne. mpi_proc_null ) then
   do j = 1, ny
    a(0,j)    = bxr(j,2)
   enddo
  endif
 if( right .ne. mpi_proc_null ) then
   do j = 1, ny
    a(nx+1,j) = bxr(j,1)
   enddo
  endif
 if( up .ne. mpi_proc_null ) then
   do i = 1, nx
    a(i,ny+1) = byr(i,1)
   enddo
  endif
 if( down .ne. mpi_proc_null ) then
   do i = 1, nx
    a(i,0)    = byr(i,2)
   enddo
  endif

  do i = 1, nx
   b(i,1)  = 0.25d0 * ( a(i-1,1)  + a(i,0)    + a(i,2)    + a(i+1,1) )
   b(i,ny) = 0.25d0 * ( a(i-1,ny) + a(i,ny-1) + a(i,ny+1) + a(i+1,ny) )
  enddo
  do j = 1, ny
   b(1,j)  = 0.25d0 * ( a(0,j)    + a(1,j-1)  + a(1,j+1)  + a(2,j) )
   b(nx,j) = 0.25d0 * ( a(nx-1,j) + a(nx,j-1) + a(nx,j+1) + a(nx+1,j) )
 enddo

  do j = 1, ny
   do i = 1, nx
    a(i,j) = b(i,j)
   enddo
  enddo
enddo
```

Computes the part that depends on the communication.

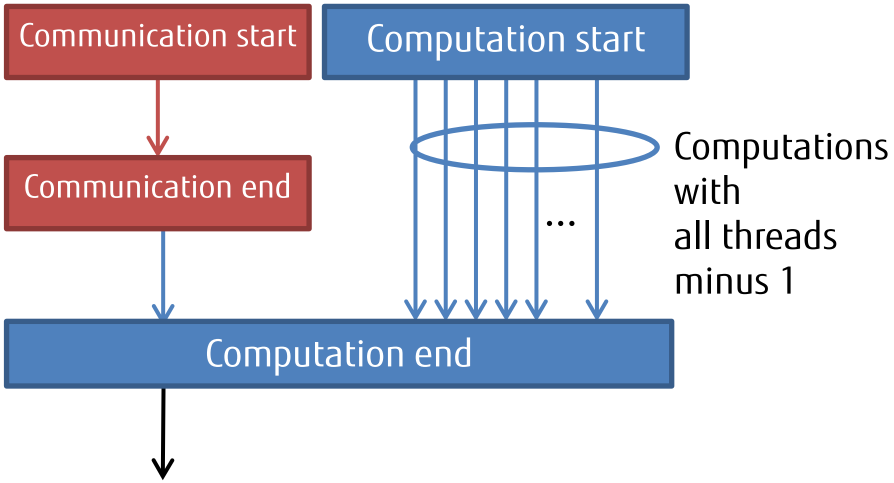
Current issue and idea for improvement

- The other threads are waiting when a thread is used for communication.
- Consider an implementation that uses one dedicated thread for communication and performs computations with the other threads.
  - MASTER + DO DYNAMIC
  - SINGLE + DO DYNAMIC

Conceptual diagram of implementation with conventional thread parallelization processing



Conceptual diagram of implementation using communication-dedicated thread



Coding Example of a Communication-dedicated Thread Using OpenMP

FUJITSU

MASTER + DO DYNAMIC

```
program main
    :
    call MPI_Init
    :
!$OMP PARALLEL
!$OMP MASTER
        MPI communication to be overlapped
        (synchronous or asynchronous allowed)
        Call MPI_Waitall
!$OMP END MASTER
!$OMP DO SCHEDULE (DYNAMIC)
    do i = 1, n
        Computation to be overlapped
    enddo
!$OMP END DO
!$OMP END PARALLEL
        Computation using communication results
    :
    call MPI_Finalize
    :
    stop
    end
```

Callouts:
- Communication by master thread. If communication ends early, thread can participate in computations
- Required for asynchronous communication
- END MASTER not synchronized
- Computation with thread other than master. Chunk size: 1
- Wait for end of overlapped communication and computation

SINGLE + DO DYNAMIC

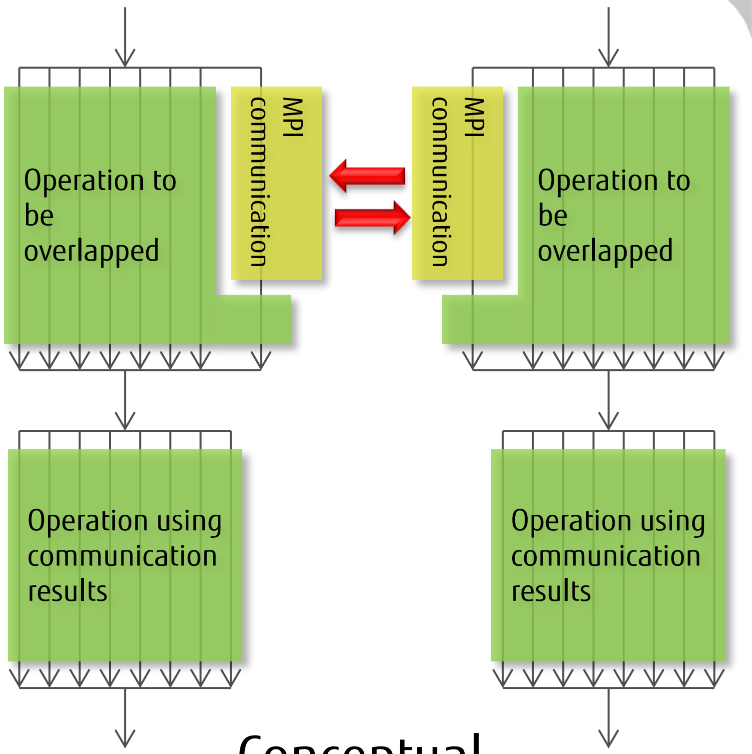
```
program main
    :
    call MPI_Init
    :
!$OMP PARALLEL
!$OMP SINGLE
        MPI communication to be
        overlapped (synchronous or
        asynchronous allowed)
        Call MPI_Waitall
!$OMP END SINGLE NOWAIT
!$OMP DO SCHEDULE (DYNAMIC)
    do i = 1, n
        Computation to be overlapped
    enddo
!$OMP END DO
!$OMP END PARALLEL
        Computation using communication results
    :
    call MPI_Finalize
    :
    stop
    end
```

Callouts:
- Communication by any single thread. If communication ends early, thread can participate in computations
- Required for asynchronous communication
- Computation with another thread. Chunk size: 1
- Wait for end of overlapped communication and computation

Explanation of coding example

FUJITSU

- Overlapping is achieved by using the MASTER/SINGLE construct with one thread as a communication-dedicated thread and by combining it with the LOOP construct having the SCHEDULE(DYNAMIC) clause .
  - One feature is that the communication-dedicated thread can also participate in operations after the end of communication.
    - Cores are not wastefully occupied by communication.
  - From the perspective of comprehensibility at the operation verification/debug time, we recommend the MASTER construct.



Conceptual diagram of operations

Improvement through the Data Types Used

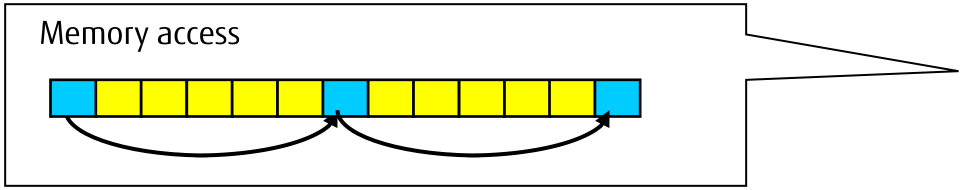
FUJITSU

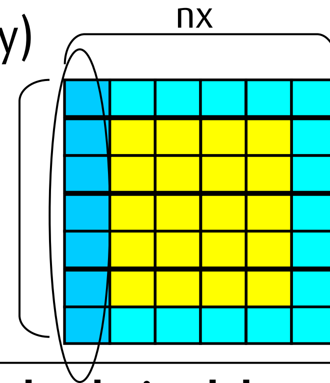
- The MPI data types include the basic datatypes and derived datatypes.
  - Basic datatypes
    - They are general data types provided in the MPI standard.

      types; e.g. MPI_INTEGER and MPI_REAL
  - Derived datatypes
    - This data type is user-defined, based on MPI basic datatypes. You can use it for point-to-point communication and collective communication in a similar way to the basic datatypes.
    - For example, use the derived datatypes to enable the MPI library to handle a (cumbersome) processing such as data packing/unpacking during communication of non-contiguous data.

Communication Using the Basic/Derived Datatype

- Communication in a non-contiguous area

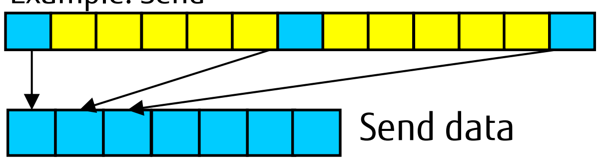




1. Using the basic datatypes

The pack/unpack should be executed explicitly by application program for send/receive data in communication.

Example: Send



Packing and sending

```
!$omp parallel do
do j=1, ny
  pack(j,1) = a(1, j)
enddo
!$omp end parallel do
mpi_isend(pack(1,1), mpi_real8,...)
```

Compiler optimization and thread parallelization can accelerate the pack/unpack.

2. Using the derived datatype

Define a derived datatype corresponding to non-contiguous data and use the datatype for communication instead of executing packing and unpacking.

```
/*  Creates derived datatype*/
mpi_type_vector(y,1,nx,mpi_real8,
                newtype,err)
/*  Registers derived datatype*/
mpi_type_commit(newtype,err)

/*  Uses derived datatype*/
mpi_isend(...,newtype,...)
```

The pack/unpack cannot be accelerated because it is handled by processing within MPI functions.

Communication Using the Basic/Derived Datatypes

FUJITSU

Using the basic datatypes (pack/unpack of send/receive data)

```
do j = 1, ny
  bxs(j,1) = a(1,j)
  bxs(j,2) = a(nx,j)
enddo

do i = 1, nx
  bys(i,1) = a(i,1)
  bys(i,2) = a(i,ny)
enddo

call mpi_irecv( bxr(1,2), ny, mpi_real8, left, 1, comm2d, req(1), err )
call mpi_irecv( bxr(1,1), ny, mpi_real8, right, 2, comm2d,req(2), err )
call mpi_irecv( byr(1,2), nx, mpi_real8, down,1, comm2d,req(3), err )
call mpi_irecv( byr(1,1) nx, mpi_real8, up,    2, comm2d, req(4), err )

call mpi_isend( bxs(1,2), ny, mpi_real8, right, 1, comm2d,req(5),err )
call mpi_isend( bxs(1,1), ny, mpi_real8, left,  2, comm2d, req(6),err )
call mpi_isend( bys(1,2), nx, mpi_real8, up,  1, comm2d, req(7),err )
call mpi_isend( bys(1,1), nx, mpi_real8, down,2,comm2d,req(8),err )

call mpi_waitall( 8, req, mpi_statuses_ignore, err )

if( left .ne. mpi_proc_null ) then
  do j = 1, ny
    a(0,j) = bxr(j,2)
  enddo
endif
```

Packs send data

Uses basic datatypes

Unpacks receive data

Using the derived datatypes

```
call mpi_type_vector( 1, nx, nx, mpi_real8, typex, err )
call mpi_type_commit( typex, err )

call mpi_type_vector( ny, 1, nx+2, mpi_real8, typey, err )
call mpi_type_commit( typey, err )

call mpi_irecv( a(0,1),     1, typey, left,    2, comm2d, req(1), err )
call mpi_irecv( a(nx+1,1), 1, typey, right,  1, comm2d, req(2), err )
call mpi_irecv( a(1,0),     1, typex, up,     2, comm2d, req(3), err )
call mpi_irecv( a(1,ny+1), 1, typex, down, 1, comm2d, req(4), err )

call mpi_isend( a(1,1),    1, typey, left,    1, comm2d, req(5), err )
call mpi_isend( a(nx,1),   1, typey, right,  2, comm2d, req(6), err )
call mpi_isend( a(1,1),    1, typex, up,     1, comm2d, req(7), err )
call mpi_isend( a(1,ny),   1, typex, down, 2, comm2d, req(8), err )

call mpi_waitall( 8, req, mpi_statuses_ignore, err )
```

Defines derived datatype

Uses derived datatype

Use of Assistant Cores (1/3)

FUJITSU

- Assistant core
  - Two assistant cores are installed for thirty-two computation cores.
  - The cores are not used by user applications but are responsible for OS processing, etc.
  - Purposes of use
    - OS noise reduction
    - **Overlapping execution of computation and communication**
    - Routing of IO data (between Tofu and InfiniBand)
- Facilitation of asynchronous communication using assistant cores
  - Use the MCA parameter opal_progress_thread_mode (specifying the computation mode for the MPI asynchronous processing progress thread) and MPI Asynchronous Communication Promotion Section Specifying Interface (FJMPI_Progress_start and FJMPI_Progress_stop) to facilitate asynchronous communication for the user-specified section by using assistant cores.
  - Communications using assistant cores improve performance when the communication method is Rendezvous.
  - This is effective in cases where the assistant cores perform most of the nonblocking communication. However, if the time taken by computations differs significantly from the time taken by communications, the effect of overlapping is reduced. Be careful when using assistant cores.

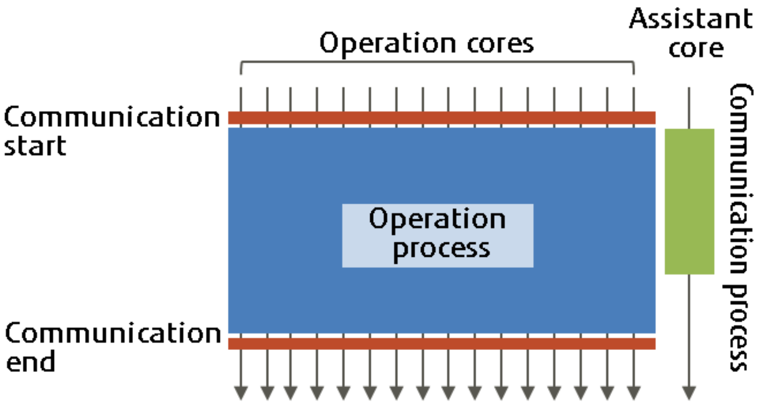
Use of Assistant Cores (2/3)

- Specify a mode in the following by MCA parameter opal_progress_thread_mode
  - 0: Specifies that the function for asynchronous communication facilitation using assistant cores not be used. The default is 0.
  - 1: Specifies to use manual section (without MPI call) mode to promote asynchronous communication using an assistant core.
    - This mode has the lowest performance overhead.
  - 2: Specifies to use manual section (with MPI call) mode to promote asynchronous communication using an assistant core.
    - This mode has a slightly higher overhead than the mode with no MPI calls.
  - 3: Specifies use of the automatic section mode.
    - If many MPI functions are called, the overhead is higher.
- Specifying the target section for the asynchronous communication facilitation
  - Use the following interfaces to specify the target section (valid in modes 1 and 2).
    - FJMPI_Progress_start:

      Starts the asynchronous communication facilitation.
    - FJMPI_Progress_stop:

      Stops the asynchronous communication facilitation.

<u>Conceptual diagram of operation when an assistant core is used</u>



Use of Assistant Cores (3/3)

FUJITSU

- Example of mode 1 (with the asynchronous communication facilitation)
  - Execution of the following asynchronous communication program with 1 specified for the MCA parameter opal_progress_thread_mode resulted in a performance improvement of about 58%.

```
  :
for (i = 0; i < count; i++) {

  MPI_Irecv  (rbuf[0], MSG_LEN, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv  (rbuf[1], MSG_LEN, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &reqs[1]);
  MPI_Isend (sbuf[0], MSG_LEN, MPI_DOUBLE, prev, 0, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend (sbuf[1], MSG_LEN, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &reqs[3]);

  FJMPI_Progress_start();

  for (j = 0; j < VEC_LEN; j++) {
    y[j] = a * x[j] + y[j];
  }

  FJMPI_Progress_stop();

  MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);

}
  :
```

Computation part (target section for asynchronous communication facilitation)

- Without asynchronous communication facilitation
  (opal_progress_thread_mode 0)
  - Overall time: About 27 seconds

- With asynchronous communication facilitation process
  (opal_progress_thread_mode 1)
  - Overall time: About 17 seconds

* Execution with 2 nodes and 2 processes

Example of Differences in Performance with a Specified Shape

FUJITSU

- Communication performance differences depending on the specified shape
  - The communication part (sendp and mpi_allreduce) of the HIMENO benchmark was used to check the differences.
  - No variation was found in the computation part, but performance differences were observed in the processing time of the communication part.
  - The performance differences when a shape is specified depend on the program.

■ Appearance of HIMENO source code

```
 :
subroutine jacobi(nn,gosa)
 :
   loop 1    ┐
    :        ├ Computation part
   loop 2    ┘
    :
   call sendp (ndx,ndy,ndz)   ┐
    :                         ├ Communication part
   call mpi_allreduce (....)  ┘
 :
end
```

Size	Source code divided shape	Tofu shape specification	MFLOPS	Processing time (msec): Total for computation part	Processing time (msec): Total for communication part	Processing time (msec): Overall time
12n24p16t	8x3x1	1x2x6	184,348	10.70	38.30	49.00
		1x3x4	178,917	10.70	39.80	50.49
		2x2x3	183,375	10.70	38.56	49.26
		2x3x2	193,632	10.70	35.96	46.65

Fujitsu Extended Specifications

- Rank Query Interface
- Extended RDMA interface

Rank Query Interface

Ranks and coordinates are mutually converted.

```c
#include <stdio.h>
#include <mpi.h>
#include <mpi-ext.h>

main(int argc, char *argv[])
{
 int rank, dim;
 int xsize, ysize, zsize;
 int myx, myy, myz;
 int xminus, yminus, zminus;
 int xplus, yplus, zplus;

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 FJMPI_Topology_get_dimension(&dim);

 FJMPI_Topology_get_shape(&xsize, &ysize, &zsize);

 switch(dim){
 case 3:
  FJMPI_Topology_rank2xyz(rank, &myx, &myy, &myz);
  FJMPI_Topology_xyz2rank((myx+1)%xsize, myy, myz, &xplus);
  FJMPI_Topology_xyz2rank(myx, (myy+1)%ysize, myz, &yplus);
  FJMPI_Topology_xyz2rank(myx, myy, (myz+1)%zsize, &zplus);
  FJMPI_Topology_xyz2rank((myx-1)%xsize, myy, myz, &xminus);
  FJMPI_Topology_xyz2rank(myx, (myy-1)%ysize, myz, &yminus);
  FJMPI_Topology_xyz2rank(myx, myy, (myz-1)%zsize, &zminus);
  printf("myrank=%2d: %2d %2d %2d %2d %2d %2d¥n",
   rank, xplus, xminus, yplus, yminus, zplus, zminus);
  break;
 case 2:
  FJMPI_Topology_rank2xy(rank, &myx, &myy);
  FJMPI_Topology_xy2rank((myx+1)%xsize, myy, &xplus);
  FJMPI_Topology_xy2rank(myx, (myy+1)%ysize, &yplus);
  FJMPI_Topology_xy2rank((myx-1)%xsize, myy, &xminus);
  FJMPI_Topology_xy2rank(myx, (myy-1)%ysize, &yminus);
  printf("myrank=%2d: %2d %2d %2d %2d¥n",
   rank, xplus, xminus, yplus, yminus);
  break;
 case 1:
  FJMPI_Topology_rank2x(rank, &myx);
  FJMPI_Topology_x2rank((myx+1)%xsize, &xplus);
  FJMPI_Topology_x2rank((myx-1)%xsize, &xminus);
  printf("myrank=%2d: %2d %2d¥n", rank, xplus, xminus);
  break;
 }

 MPI_Finalize();
}
```

Number of dimensions

Job shape

Rank calculated from adjacent coordinates in 3 dimensions

Note

- Ranks cannot be obtained from a dynamically generated MPI process.

Extended RDMA Interface

FUJITSU

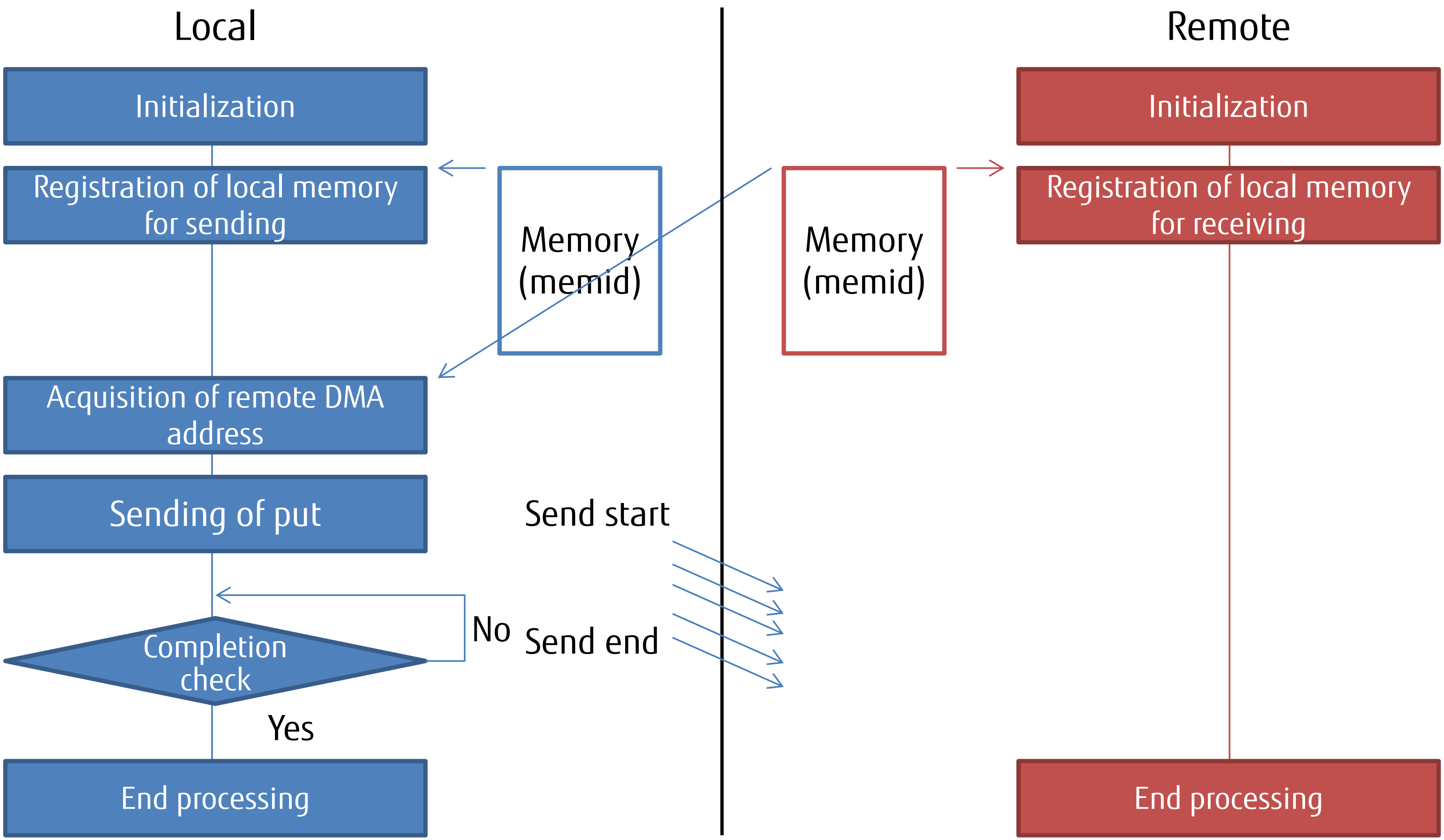
- What is the extended RDMA interface?
  - Communication through this interface can make the most of Tofu characteristics, such as communication using four network interfaces and communication using alternative paths.
- API

API	Function overview
FJMPI_Rdma_init	Initialization of extended RDMA interface
FJMPI_Rdma_finalize	End processing of extended RDMA interface
FJMPI_Rdma_reg_mem	Memory registration
FJMPI_Rdma_dereg_mem	Release of memory registration
FJMPI_Rdma_get_remote_addr	Acquisition of remote DMA address
FJMPI_Rdma_put	RDMA Write communication (put)
FJMPI_Rdma_get	RDMA Read communication (get)
FJMPI_Rdma_armw	RDMA ARMW communication (atomic read modify write)
FJMPI_Rdma_poll_cq	RDMA completion check
FJMPI_Rdma_poll_cq_ret_data	Acquisition of data associated with RDMA completion check and communication

- Restrictions
  - The available memory IDs for identifying communication areas are 0 to 510.
  - The available message tag numbers for identifying transfer data are 0 to 14.

How to Use the Extended RDMA Interface

Processing flow (RDMA Write method)



Sample Program for the Extended RDMA Interface

FUJITSU

PingPong communication proceeds between even and odd ranks.

```c
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <mpi-ext.h>

#define MEMID1 10
#define MEMID2 11
#define BUFSIZE 1024

int main(int argc, char *argv[])
{
 int      i, lrank, rrank, size;
 uint64_t  laddr1, raddr2;
 struct FJMPI_Rdma_cq  cq;
 volatile long   *sbuf = malloc(BUFSIZE*sizeof(long));
 volatile long   *rbuf = malloc(BUFSIZE*sizeof(long));
 double    d1, d2;

 MPI_Init(&argc, &argv);
 FJMPI_Rdma_init();

 MPI_Comm_rank(MPI_COMM_WORLD, &lrank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 if(size % 2){
  fprintf(stderr, "MPI_Comm_size ERROR¥n");
  free((void *)sbuf); free((void *)rbuf);
  MPI_Abort(MPI_COMM_WORLD, -1);
 }

 rrank = (lrank % 2) ? lrank - 1 : lrank + 1;

 for(i = 0; i < BUFSIZE; ++i){ sbuf[i] = i; rbuf[i] = -1;}

 laddr1 = FJMPI_Rdma_reg_mem(MEMID1, (void *)sbuf, BUFSIZE*sizeof(long));
 FJMPI_Rdma_reg_mem(MEMID2, (void *)rbuf, BUFSIZE*sizeof(long));
 while((raddr2 = FJMPI_Rdma_get_remote_addr(rrank, MEMID2)) == FJMPI_RDMA_ERROR);
```

Address exchange

```c
 MPI_Barrier(MPI_COMM_WORLD);

 if((lrank % 2) == 0){
  d1 = MPI_Wtime();
  for(i = 0; i < BUFSIZE; ++i){
   if(FJMPI_Rdma_put(rrank, i%14, raddr2+i*sizeof(long), laddr1+i*sizeof(long), sizeof(long),
    FJMPI_RDMA_LOCAL_NIC0 | FJMPI_RDMA_REMOTE_NIC0 | FJMPI_RDMA_STRONG_ORDER)){
    fprintf(stderr, "FJMPI_Rdma_put ERROR¥n");
    exit(EXIT_FAILURE);
   }
   while(FJMPI_Rdma_poll_cq(FJMPI_RDMA_NIC0, &cq) != FJMPI_RDMA_NOTICE);
   if((cq.pid != rrank) || (cq.tag != (i%14))){
    fprintf(stderr, "FJMPI_Rdma_poll_cq ERROR¥n");
    exit(EXIT_FAILURE);
   }
   while(rbuf[i] != i);
  }
  d2 = MPI_Wtime();
  fprintf(stdout, "[%3d] Pingpong = %7.3f us¥n", lrank, (d2 - d1) * 1.0e6 / BUFSIZE / 2);
```

even rank: Receiving after Put

```c
 }else{
  for(i = 0; i < BUFSIZE; ++i){
   while(rbuf[i] != i);
   if(FJMPI_Rdma_put(rrank, i%14, raddr2+i*sizeof(long), laddr1+i*sizeof(long), sizeof(long),
     FJMPI_RDMA_LOCAL_NIC0 | FJMPI_RDMA_REMOTE_NIC0 | FJMPI_RDMA_STRONG_ORDER)){
    fprintf(stderr, "FJMPI_Rdma_put ERROR¥n");
    exit(EXIT_FAILURE);
   }
   while(FJMPI_Rdma_poll_cq(FJMPI_RDMA_NIC0, &cq)!=FJMPI_RDMA_NOTICE);
   if((cq.pid != rrank) || (cq.tag != (i%14))){
    fprintf(stderr, "CQ ERROR¥n");
    exit(EXIT_FAILURE);
   }
  }
 }
```

odd rank: Put after received

```c
 FJMPI_Rdma_finalize();
 MPI_Finalize();
 free((void*)sbuf); free((void*)rbuf);
 return 0;
}
```

Troubleshooting

- Debug library
- Memory area-related MPI errors
- Hardware Queue Overflow
- Debug options
  - Deadlock detection function
  - Communication buffer write damage detection function

Debug Library

FUJITSU

How to use the library

```
mpiexec --debuglib ...
```

What can you do with the debug library?

Operate the runtime argument check function.

- If the contents of an argument are clearly incorrect, the program ends with an error message.

  (Example) Details of the MPI_Send check

  - Is the communicator valid?
  - Is count < 0 not specified?
  - Is the tag value valid?
  - Does the rank number of the send destination exist?
  - Does datatype exist?

  etc.

Output additional information on the MPI library.

- When requesting an inspection for a problem in MPI library operation, attach the output information to the request.

Note

- The MPI program execution time may be much longer because of the linked MPI library for debugging. Take sufficient care about using the library.

Argument Check Example

FUJITSU

- Program with an incorrect argument in MPI_Comm_rank

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  MPI_Comm    comm = 0;
  int         rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(comm, &rank);

  MPI_Finalize();
}
```

invalid comm

stderr output

```
+ mpiexec --debuglib ./a.out
[mpi::mpi-errors::mpi_errors_are_fatal]
[em15-020:6039] *** An error occurred in MPI_Comm_rank
[em15-020:6039] *** on communicator MPI_COMM_WORLD
[em15-020:6039] *** MPI_ERR_COMM: invalid communicator
[em15-020:6039] *** MPI_ERRORS_ARE_FATAL (your MPI job will now abort)
[ERR.] PLE 0019 plexec One of MPI processes was
aborted.(rank=0)(nid=0x01010024)(CODE=1783,7940508049066557 44,1280)
```

Memory area-related MPI Errors (1)

- If an invalid address is used for MPI communication, the following error occurs internally in MPI.

```
[mpi::common-tofu::tofu-stag-error] Failed to query/register Tofu STag.
 [Where: btl:prepare_src, RC: -1, TNI: 0, Addr: (nil), Size: 400000]
```

  - Any occurrence of this error is likely due to insufficient memory.
    - Reduce the memory used, and try again.

- We recommend you run an error check when allocating memory.

  Be careful when using the Fortran STAT specifier.

C program

```
sbuf = malloc(SIZE);
if(sbuf == NULL){
   error;
}
```

Fortran program

```
ALLOCATE(SBUF(N,M), STAT=IERR)
IF(IERR.ne.0) THEN
 reallocate or error
ENDIF
```

Memory area-related MPI Errors (2)

FUJITSU

- Execution may end abnormally with the following error during a collective communication procedure.

```
[q20-062:5045] *** An error occurred in MPI_Gather
[q20-062:5045] *** on communicator MPI COMMUNICATOR 3 SPLIT FROM 0
[q20-062:5045] *** MPI_ERR_INTERN: internal error
[q20-062:5045] *** MPI_ERRORS_ARE_FATAL (your MPI job will now abort)
```

- MPI_ERR_INTERN that occurs during collective communication is output when the work buffer could not be acquired during collective communication.
  - Reduce the memory used, and try again.

Hardware Queue Overflow

FUJITSU

- Successive communications that use nonblocking communication or the Eager communication method may cause a runtime error.

```
[mpi::common-tofu::tofu-signal-mrq] Tofu interconnect detected
MRQ overflow. [signo:34 cq:4]
```

- Corrective action
  - Increase the number of entries in the completion queue by using the MCA parameter common_tofu_num_mrq_entries.
  - Review the communication logic.
    - In a pattern that concentrates sends at a specific node, a runtime error may occur because the receiver processing cannot keep up.
    - For successive nonblocking sends, insert MPI_Wait, MPI_Test, or other such function to facilitate the operation of the receiver.

Deadlock Detection Mechanism

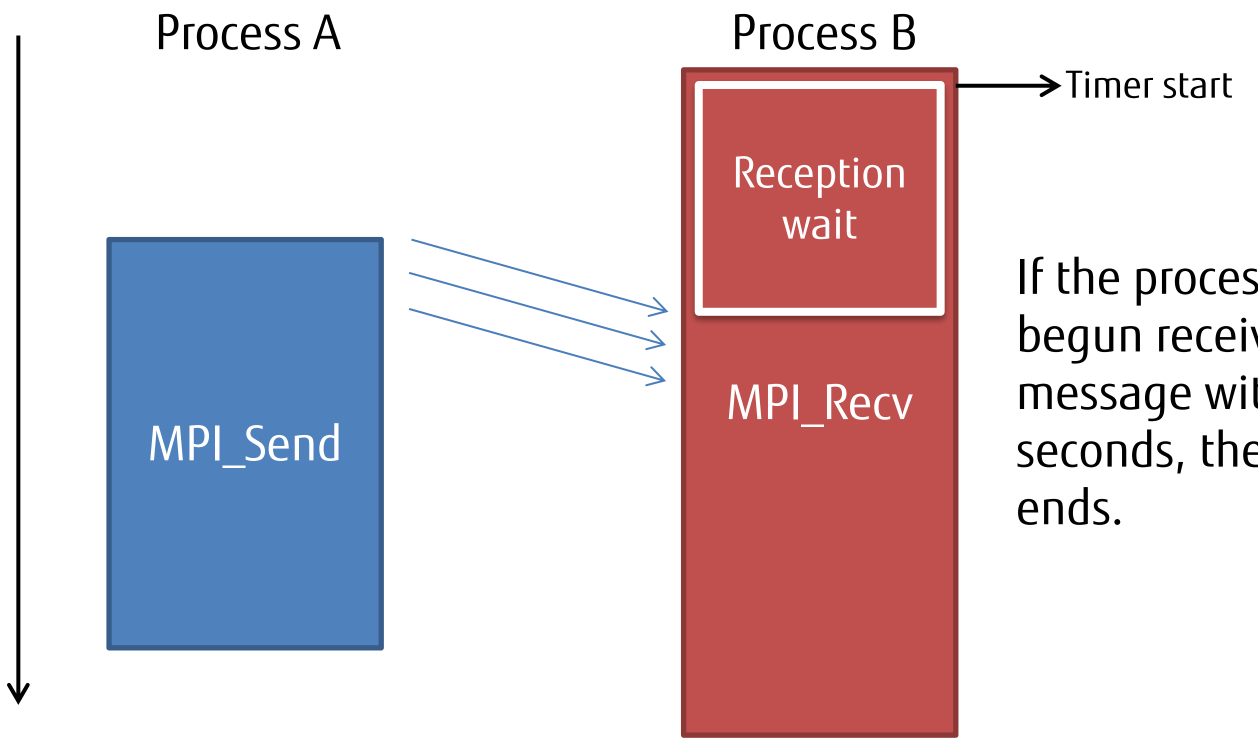
FUJITSU

How to use the mechanism

```
mpiexec --mca mpi_deadlock_timeout N ...
```

Function

- The mechanism sets a timer at the start time of receiving. Then, if the timer is exceeded, the program ends.



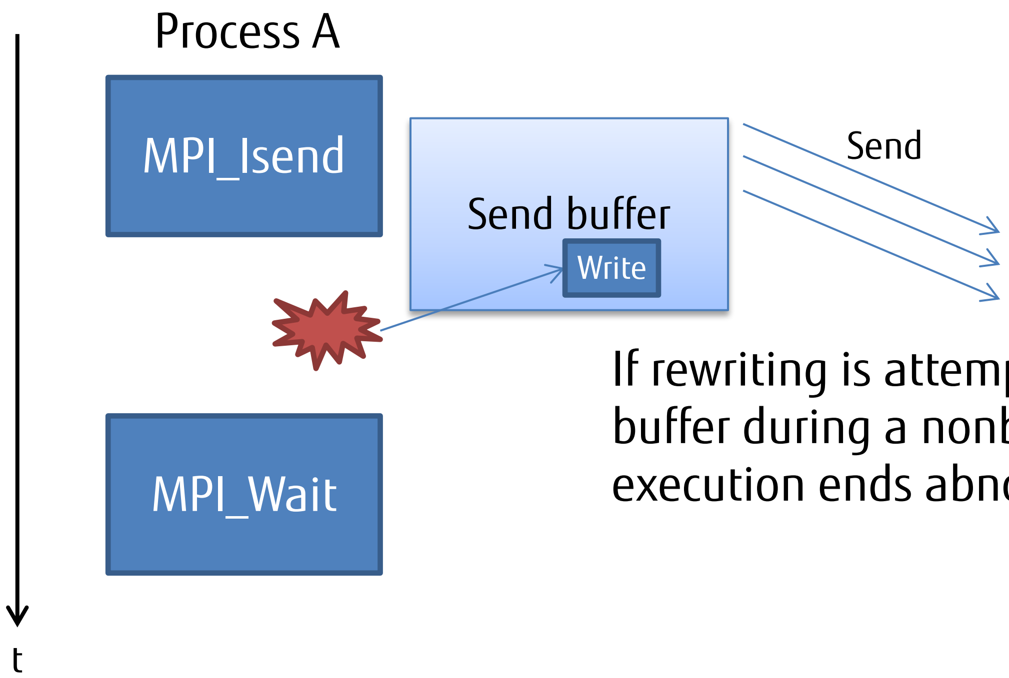
If the processing has not begun receiving a message within N seconds, the program ends.

Communication Buffer Write Damage Detection Function

How to use the function

```
mpiexec --mca mpi_check_buffer_write 1 ...
```

Function



If rewriting is attempted on the send buffer during a nonblocking send, execution ends abnormally.

Note

- MPI_Ibsend is not subject to this function.

Revision History

Version	Date	Revised section	Details
2.0	April 25, 2016	-	- First published

FUJITSU

shaping tomorrow with you