講義9：最適化

石川裕　　　　代講：堀敦史
理化学研究所AICS
東京大学

Summer School 2013

2013/08/09 9:00～10:30

Table of Contents

Performance of Parallel Program

Amdahl's Law：The performance improvement of a parallelized program is limited due to its sequential part that cannot be parallelized

Sequential Program ＝ **Sequential Code** ＋ **Parallelizable Code**

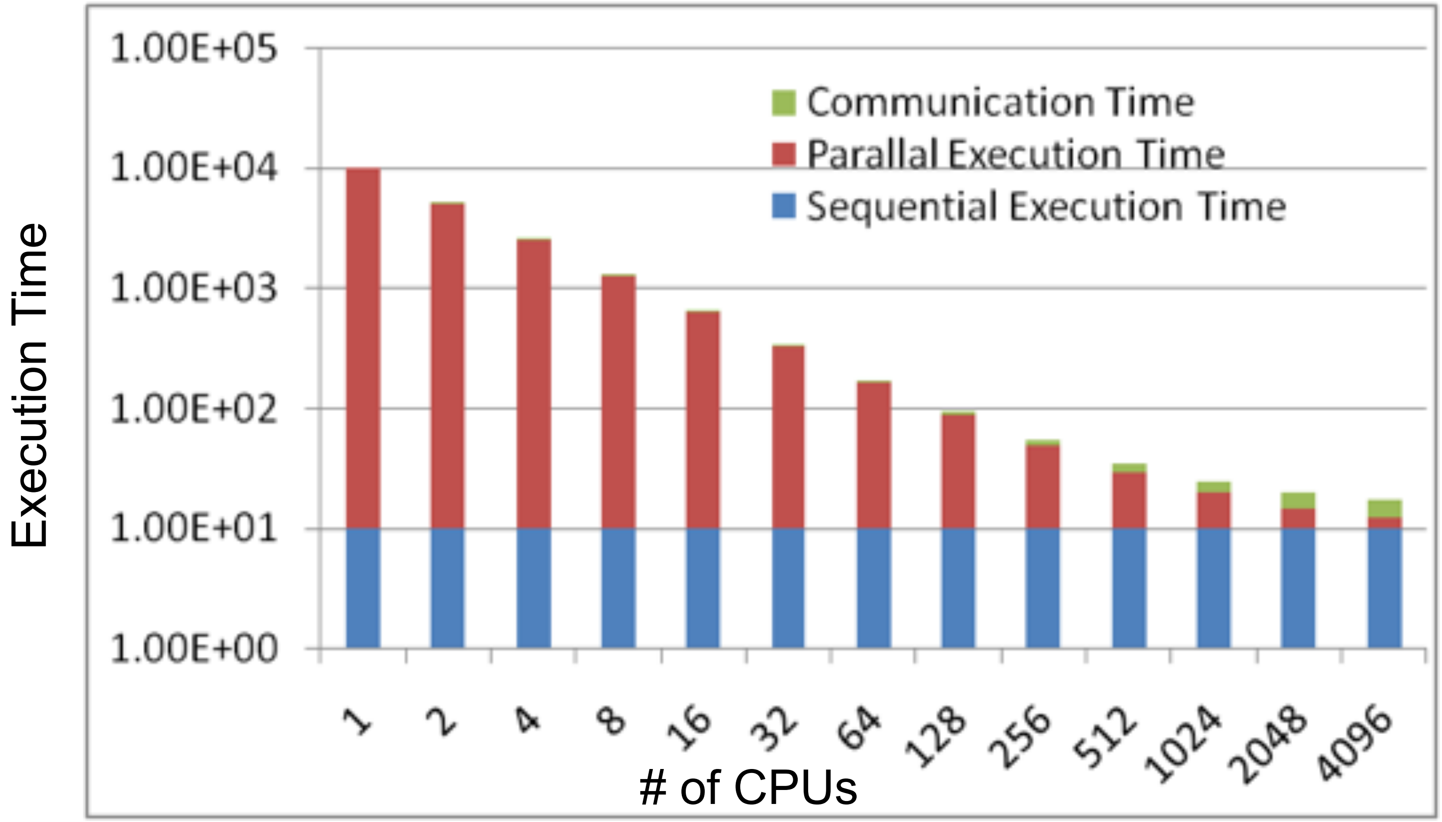
10100 ＝ 10 ＋ 10000

↓

Parallel Program ＝ **Sequential Code** ＋ **Parallelized Code** 10000 /# of CPUs ＋ **Communication**

10 ＋ 5

Performance of Parallel Program



Execution Time = Sequential Execution Time(10) + 10000/# of CPUs + Comm. Time (5)

Strong Scaling: How the performance is improved if # of CPUs is increased with fixed problem size

Performance of Parallel Program

Sequential Program = Sequential Code + Parallelizable Code

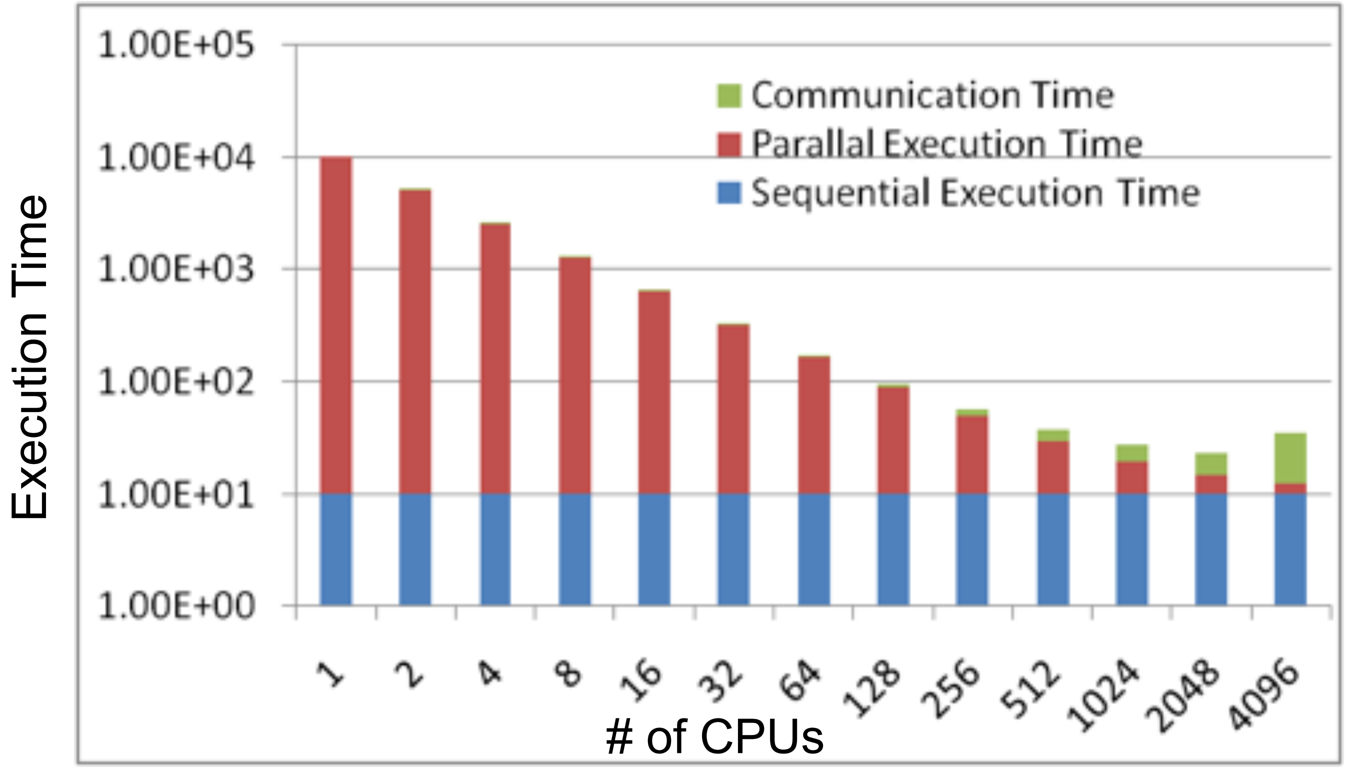
10100 = 10 + 10000

↓

Parallel Program = Sequential Code + Parallelized Code + Communication

10 + 10000 /# of CPUs + 5 + log(#ofCPUs)*5

Performance of Parallel Program



Execution Time = Sequential Execution Time(10) + 10000/# of CPUs + Comm. Time (5+log(#ofCPUs)*5)

13年8月8日木曜日

Performance of Parallel Program: Weak Scaling

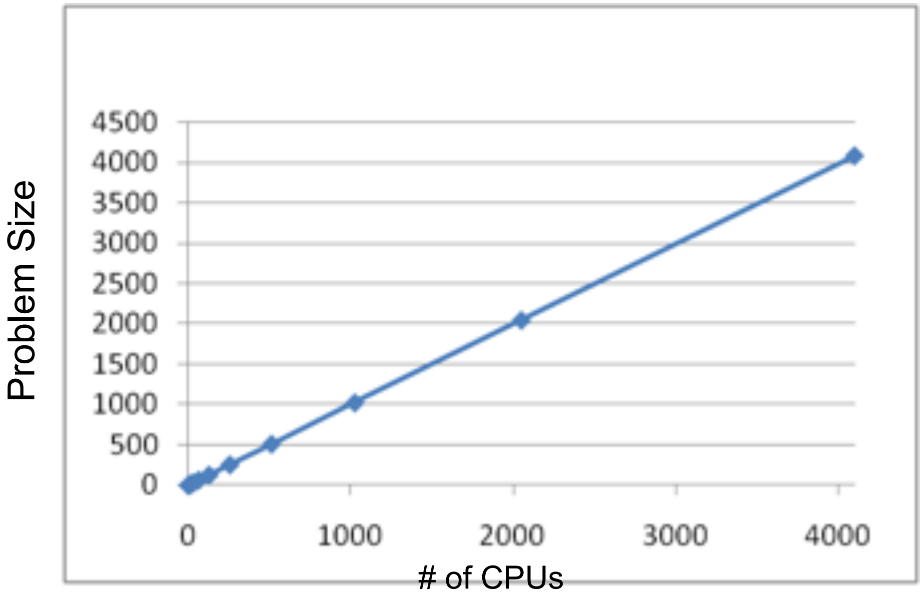
Weak Scaling: How much problem size is increased with increasing # of CPUs such that the execution time is same

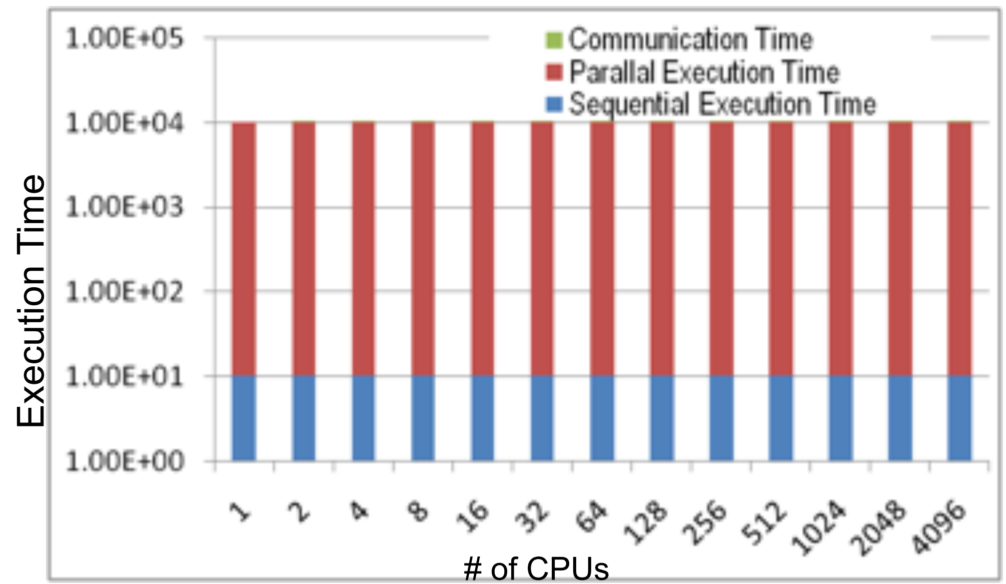
Parallel Program = **Sequential Code** + **Parallelized Code** + **Communication**

Sequential Code: 10

Parallelized Code: 10000 *# of Procs/# of CPUs

Communication: 5 + log(#ofCPUs)*5





13年8月8日木曜日

Performance of Parallel Program: Summary

- Strong Scaling
  - How the performance is improved if # of CPUs is increased with fixed problem size
  - Communication time (latency) is the key for scalability
  - Sequential execution time becomes dominant
- Weak Scaling
  - How much problem size is increased with increasing # of CPUs such that the execution time is same

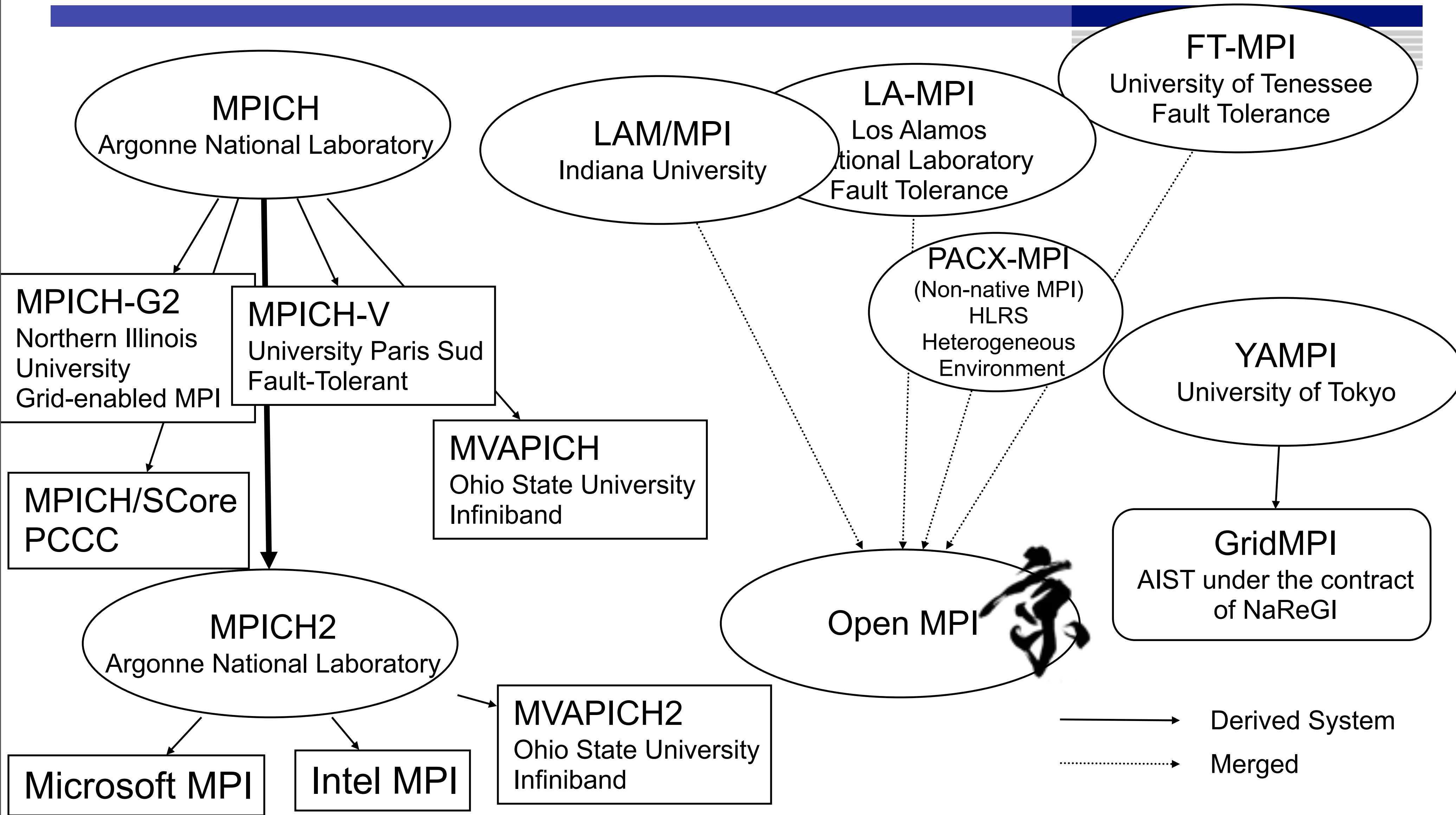
What is MPI Standard

- MPI Standard does specify API (Application Program Interface)
  - Functions such as MPI_Send and MPI_Receive
  - Symbols such as MPI_COMM_WORLD and MPI_SUCCESS
- MPI Standard does not specify runtime parameters for tuning communication performance
- MPI Standard does not specify BPI (Binary Program Interface)
  - Representations of Symbols
- MPI Standard does not specify Communication Protocol

- As a result, the executable MPI program compiled under a specific MPI implementation can only run on the same MPI implementation environment.
- Thus, it is important which MPI implementation you use in your environment.

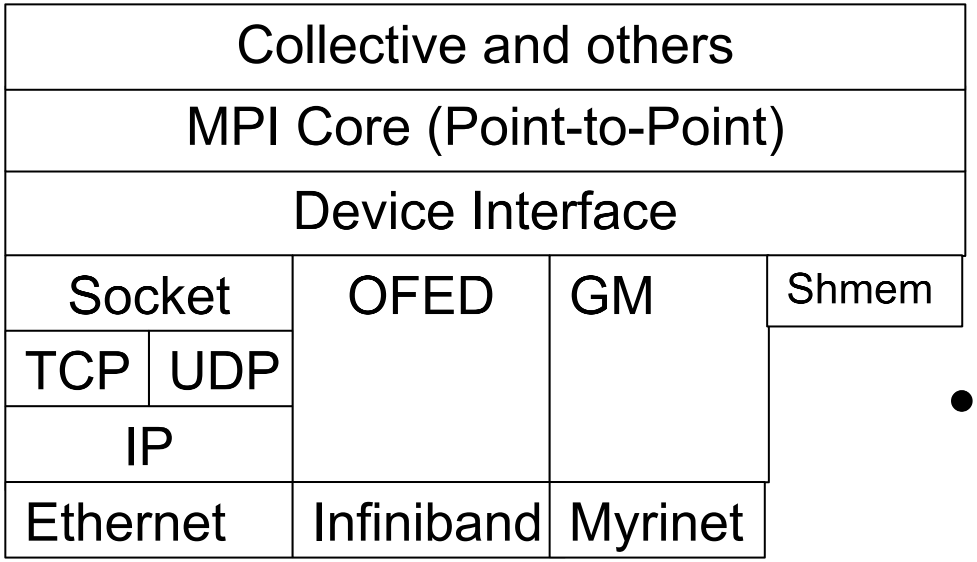
http://www.mpi-forum.org/
http://meetings.mpi-forum.org/

Various MPI Library Implementations



13年8月8日木曜日

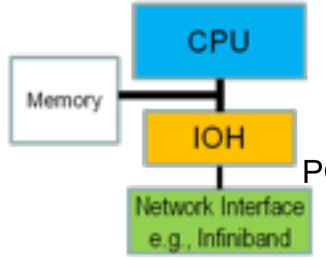
MPI Layered Implementation Structure and Network Hardware



OFED: Device driver for Infiniband (IB) network

	Throughput (Gbyte/sec)	Latency (usec)	RDMA
1/10G	0.125/1.25	10～	NO
Infiniband (IB)	4	～2	YES

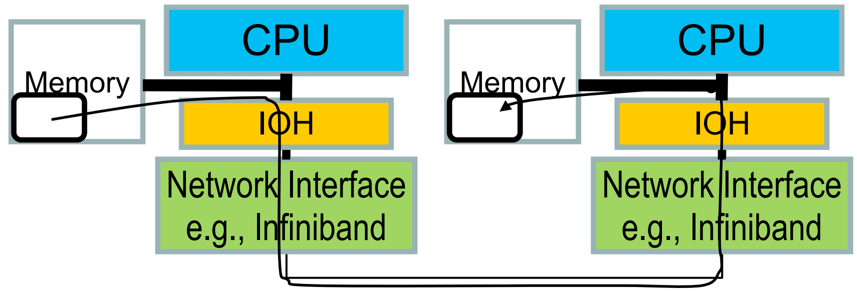
4xFDR IB 14 Gbps (8B/10B)x4 = 5.6 GByte/sec



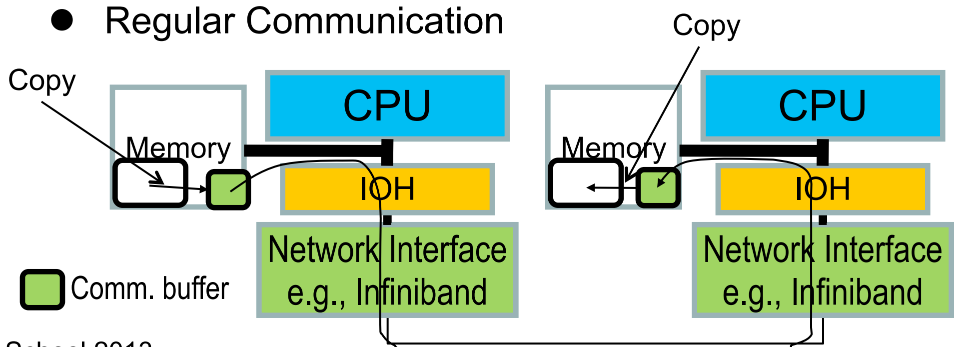
	GT/sec	encoding	Gbit/sec
Gen1	2.5	8B/10B	2
Gen2	5	8B/10B	4
Gen3	8	128B/130B	7.877

IOH: I/O Hub

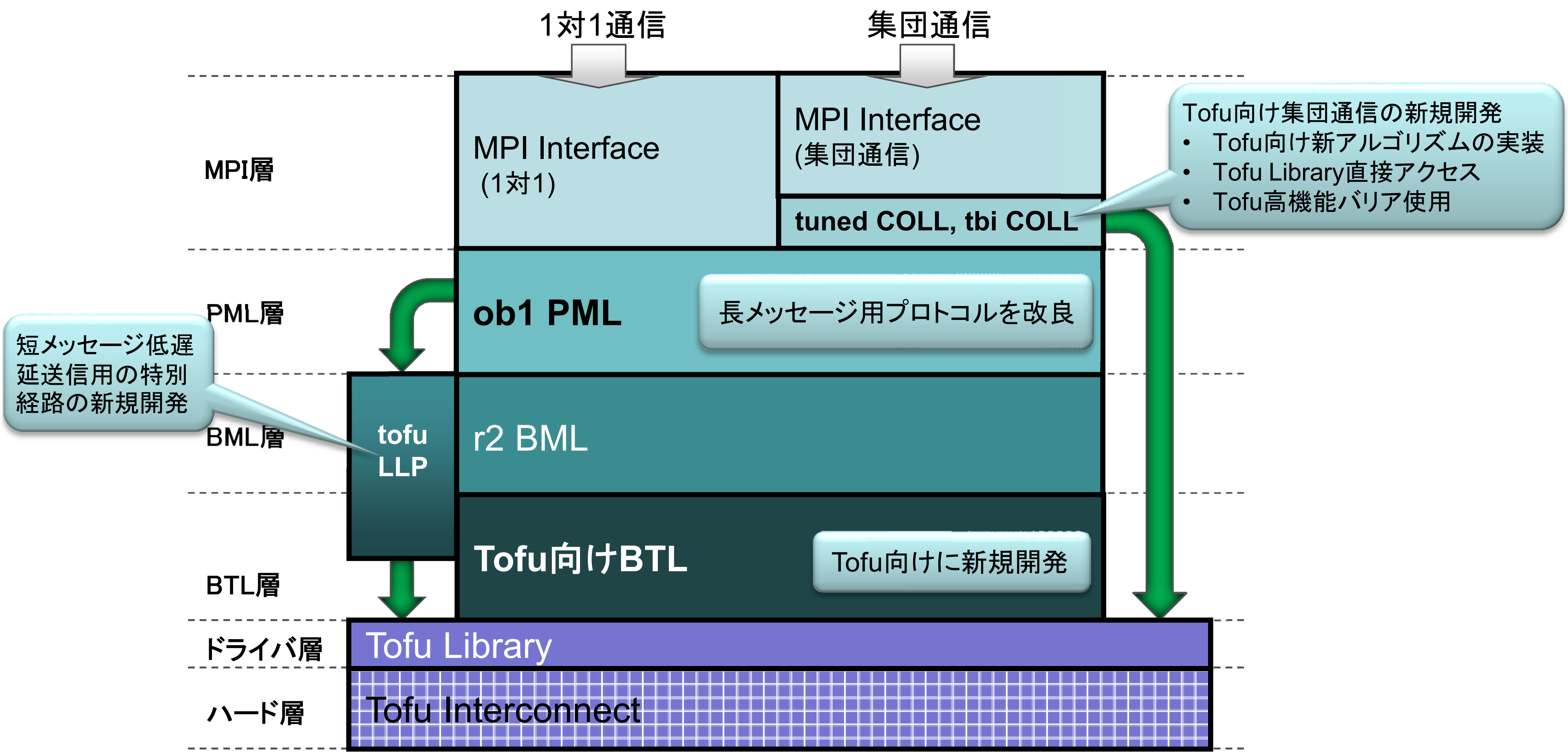
- RDMA (Remote Direct Memory Access): Remote memory is directory accessed without CPU handling



- Regular Communication



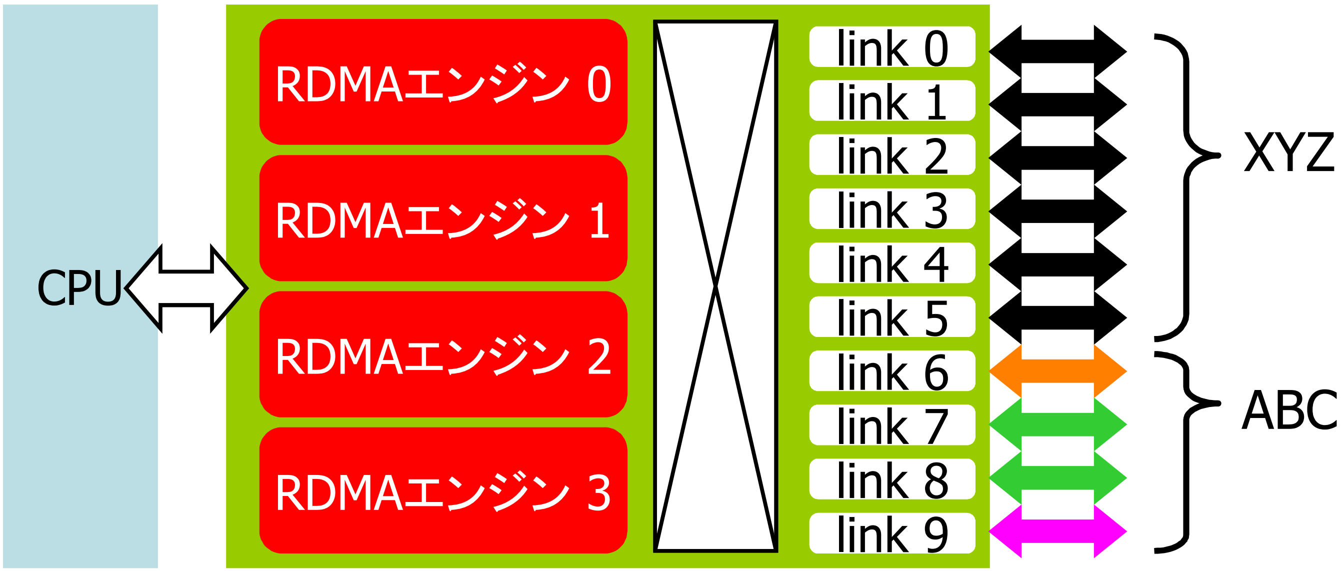
「京」の MPI（OpenMPI ベース）



FUJITSU. 63, 3, p. 299-304（05, 2012）

京の Tofu Network Interface (TNI)

- ポート数10（XYZ軸6ポート＋ABC軸4ポート）
- 4つのRDMAエンジンを搭載、同時に4送信4受信が可能

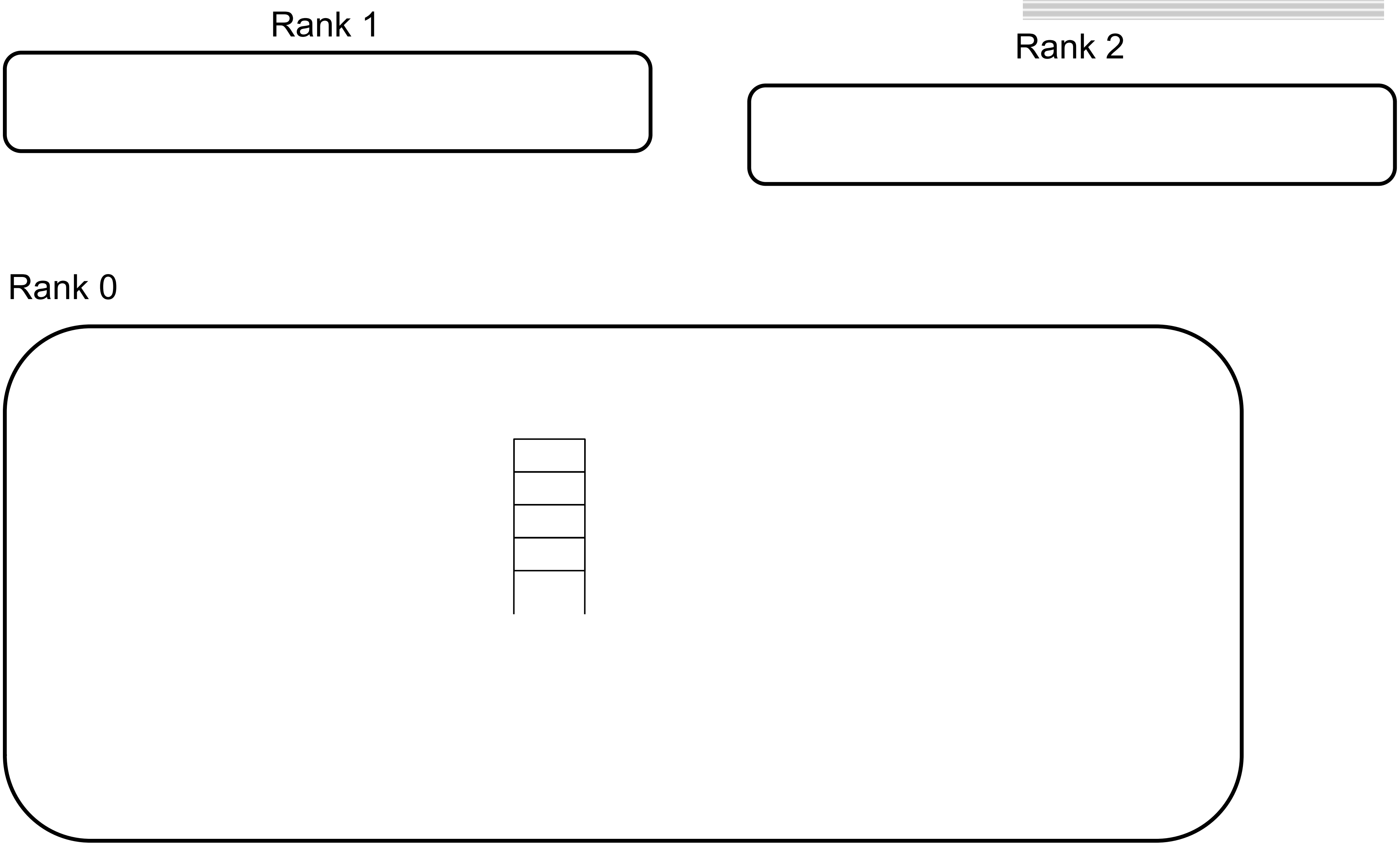


ノードあたり 理論性能	TSUBAME 2.0 InfiniBand QDR	Cray XE6 Hopper Gemini 1.2	「京」 Tofu Interconnect	IBM Blue Gene/Q 5D-Torus
演算性能	2391 GFlops	153.6 GFlops	128 GFlops	204.8 GFlops
リンク帯域（片方向）	4 GB/s	5.8 GB/s	5 GB/s	2 GB/s
同時通信数	2	1	4	10
同時通信帯域（片方向）	8 GB/s	8.3 GB/s	20 GB/s	20 GB/s

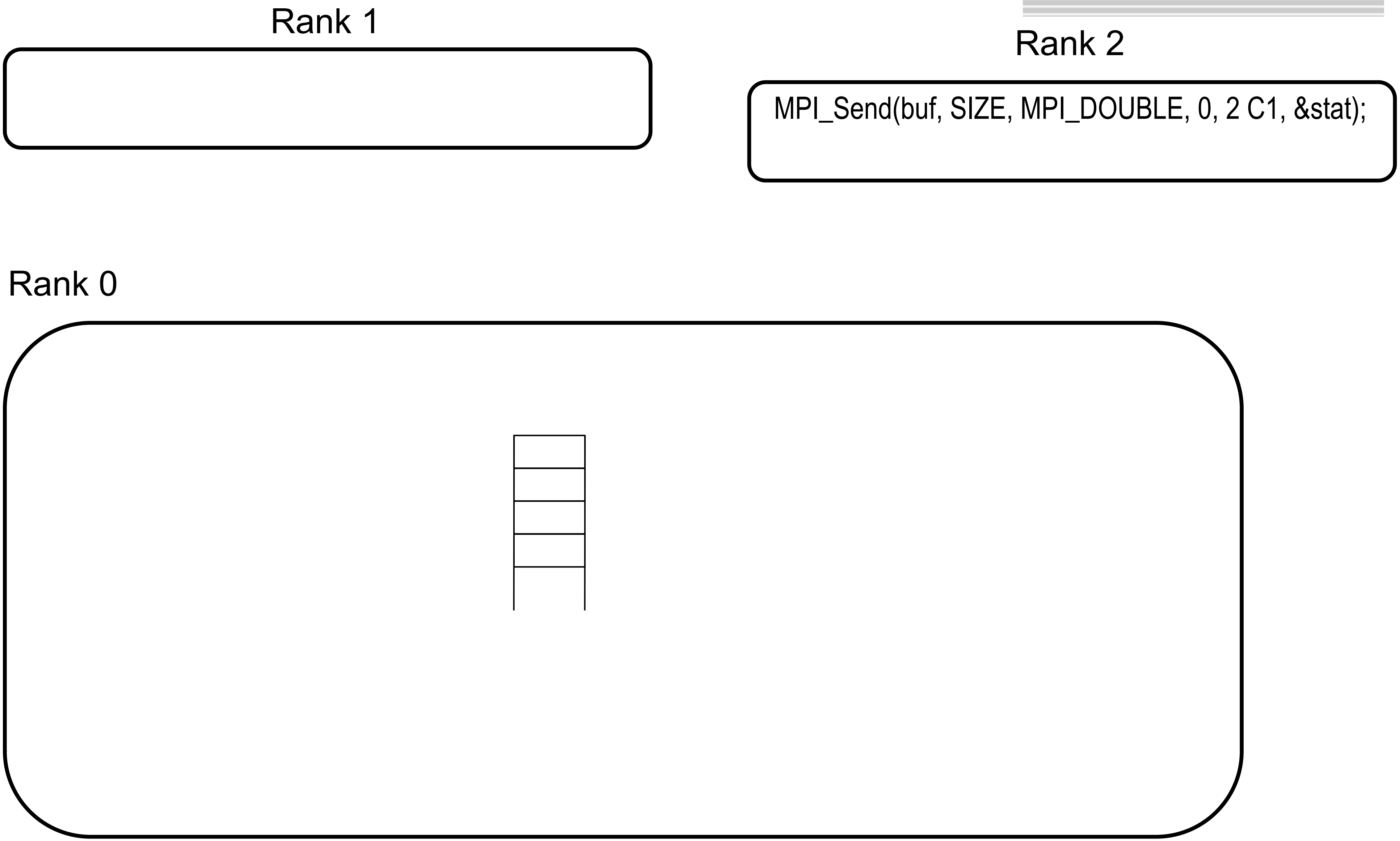
Inside MPI: Basic

- Eager Protocol
  - When a message send primitive is posted, the message is immediately sent to the receiver

Inside MPI: Basic



Inside MPI: Basic



Inside MPI: Basic

Rank 1

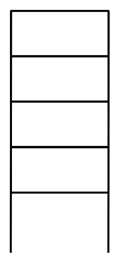
Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
```

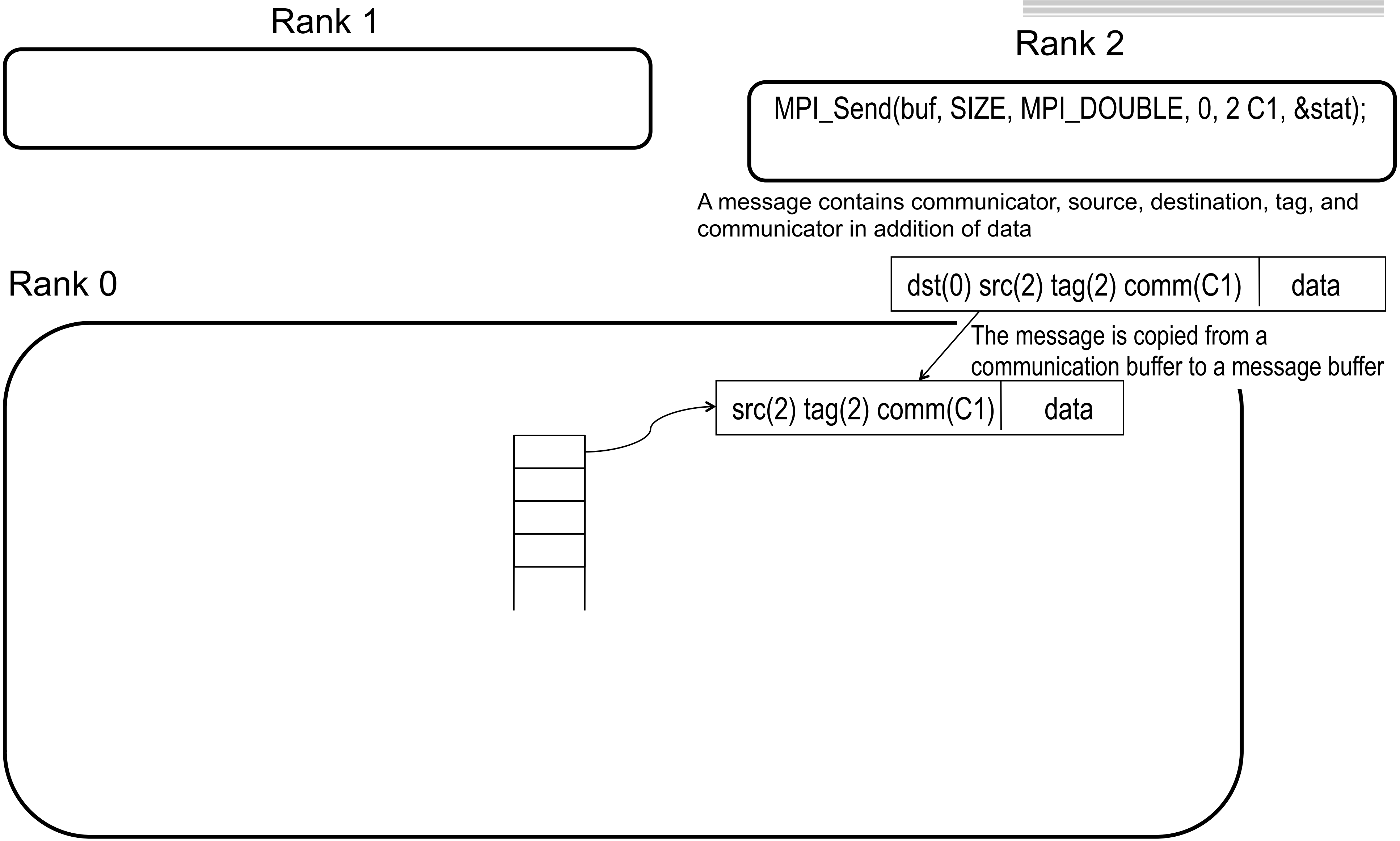
A message contains communicator, source, destination, tag, and communicator in addition of data

dst(0) src(2) tag(2) comm(C1)	data

Rank 0

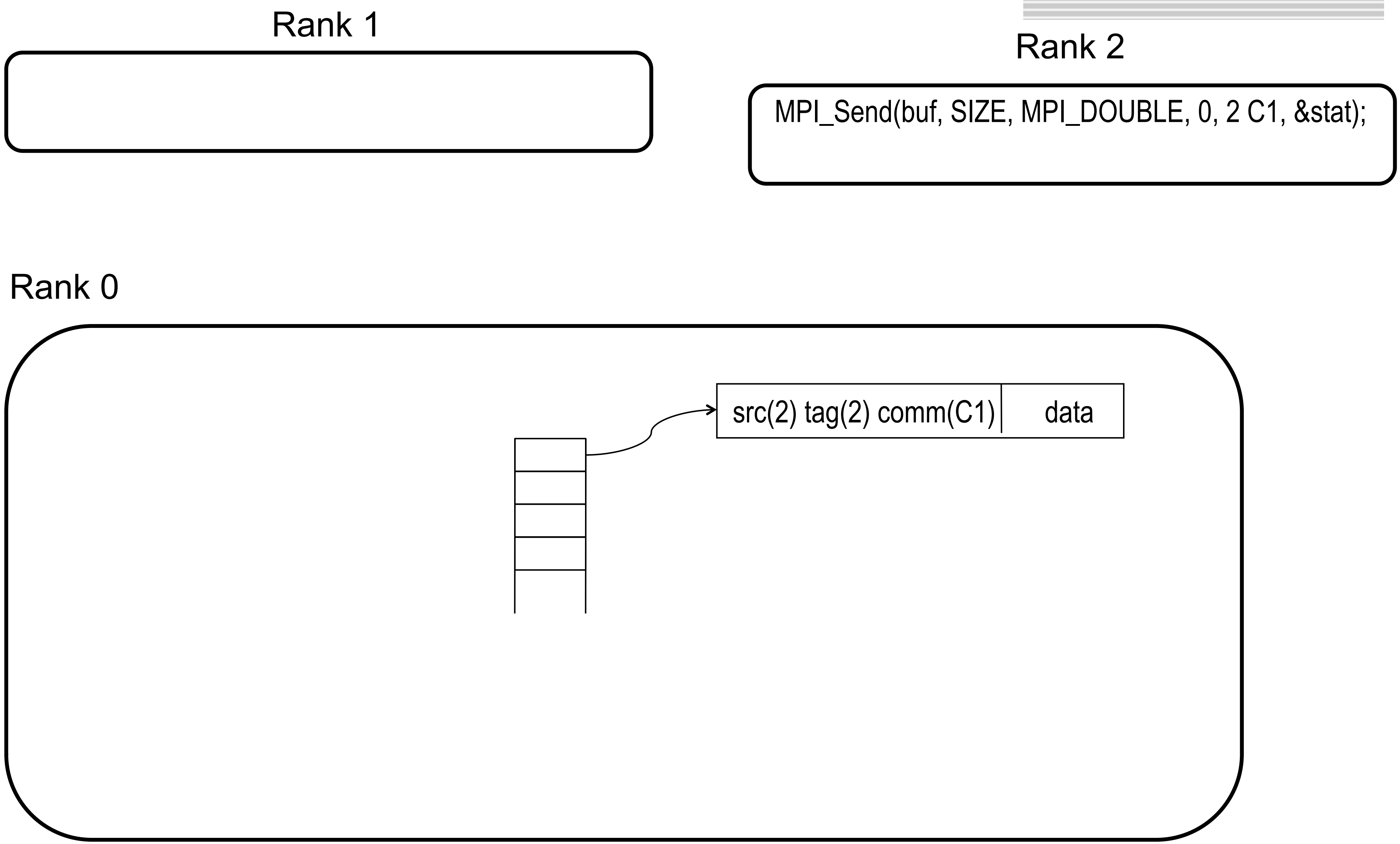


Inside MPI: Basic



13年8月8日木曜日

Inside MPI: Basic



Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
```

Rank 0

13年8月8日木曜日

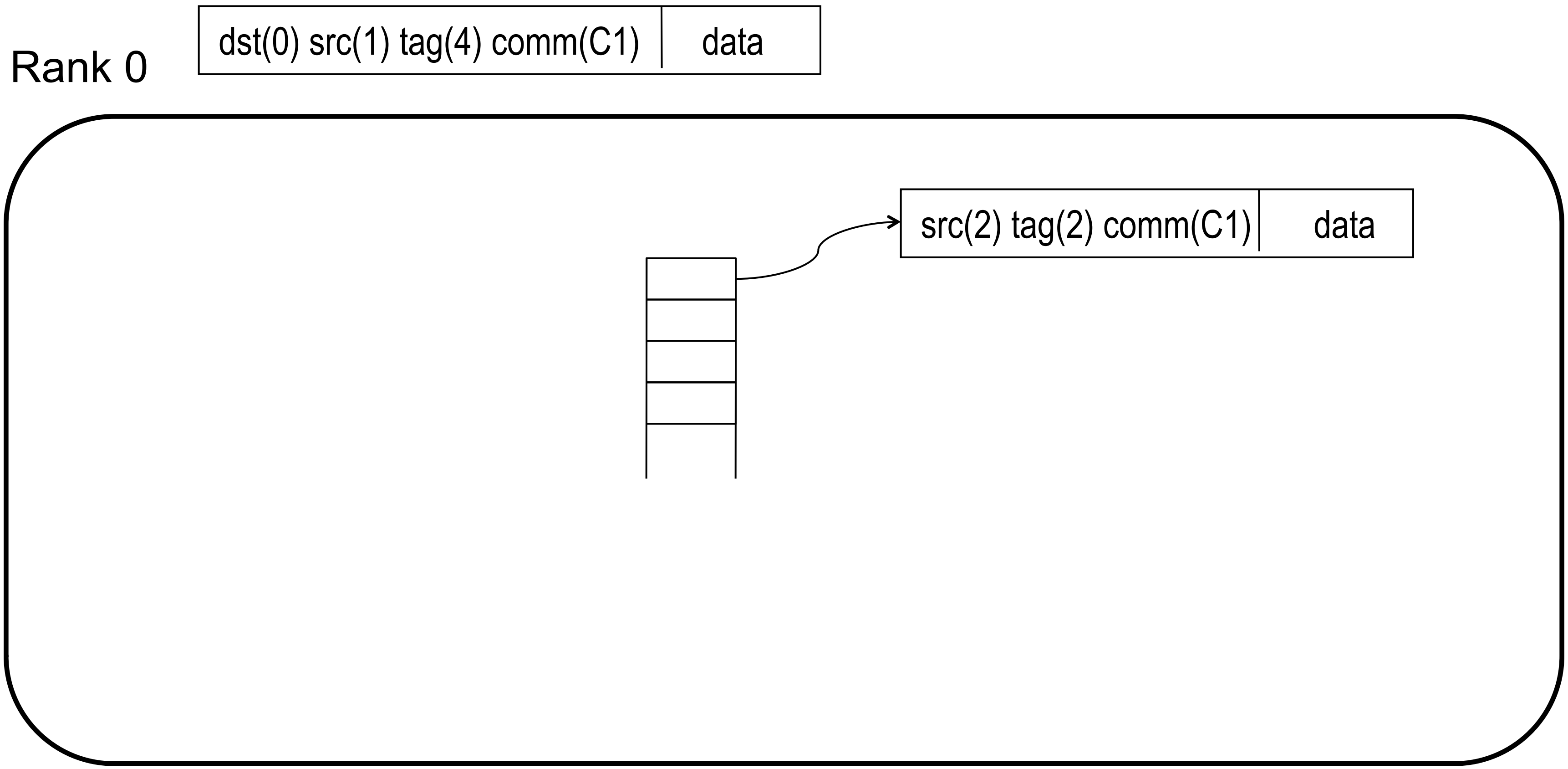
Inside MPI: Basic

Rank 1

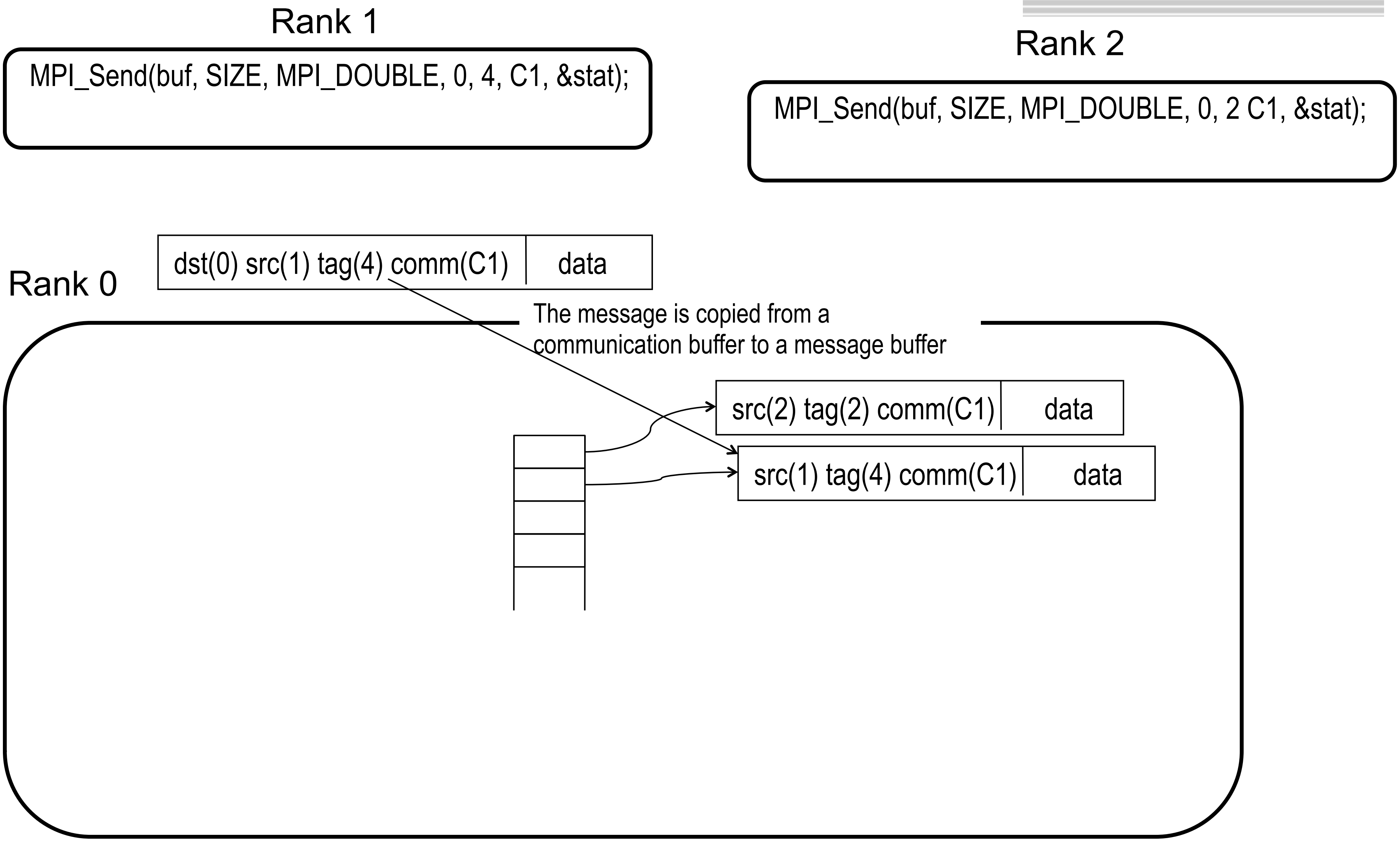
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);

Rank 2

MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);



Inside MPI: Basic



13年8月8日木曜日

Inside MPI: Basic

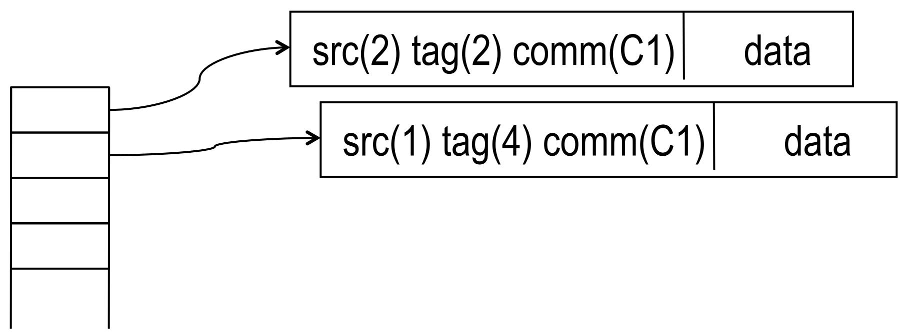
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic

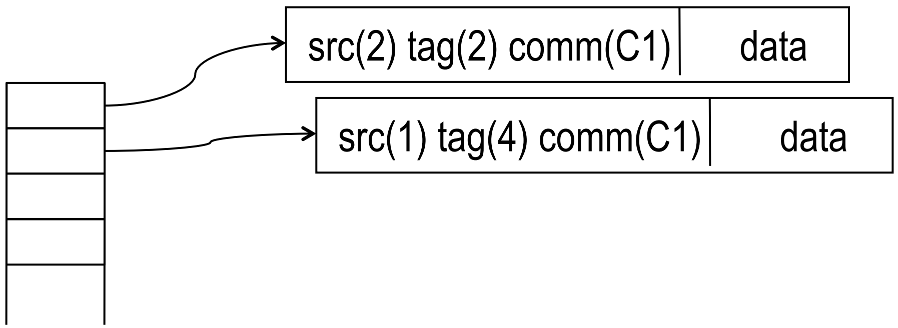
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

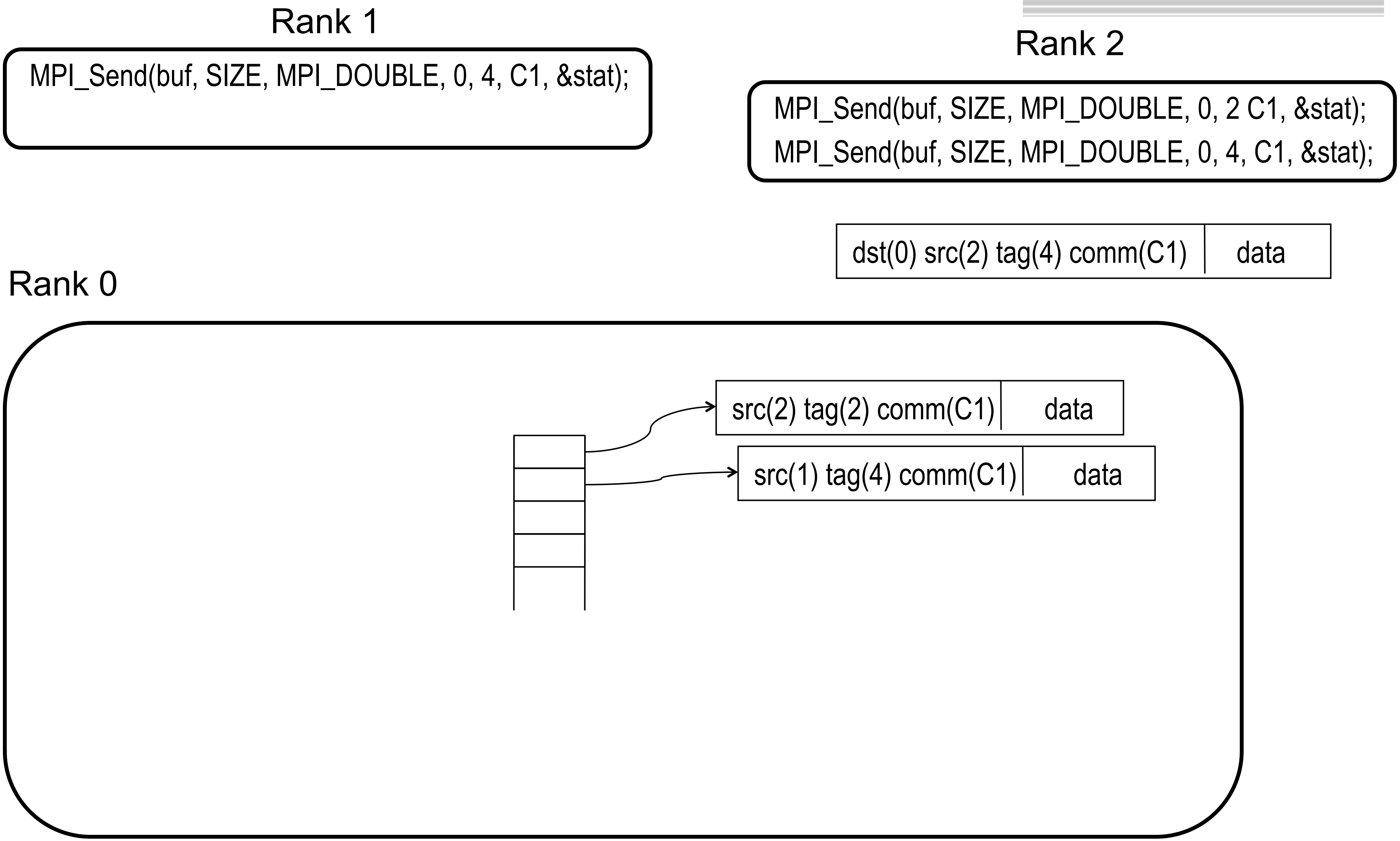
```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic



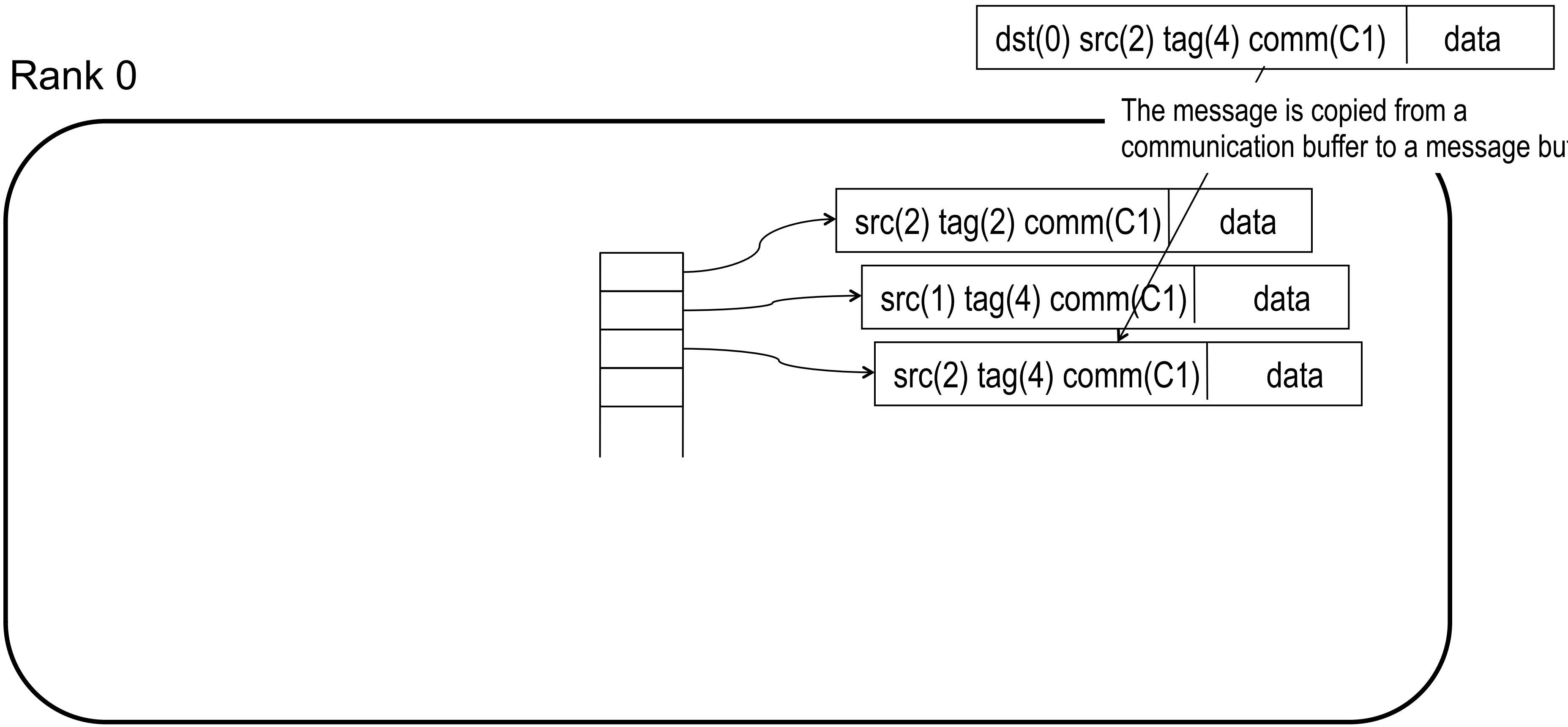
Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```



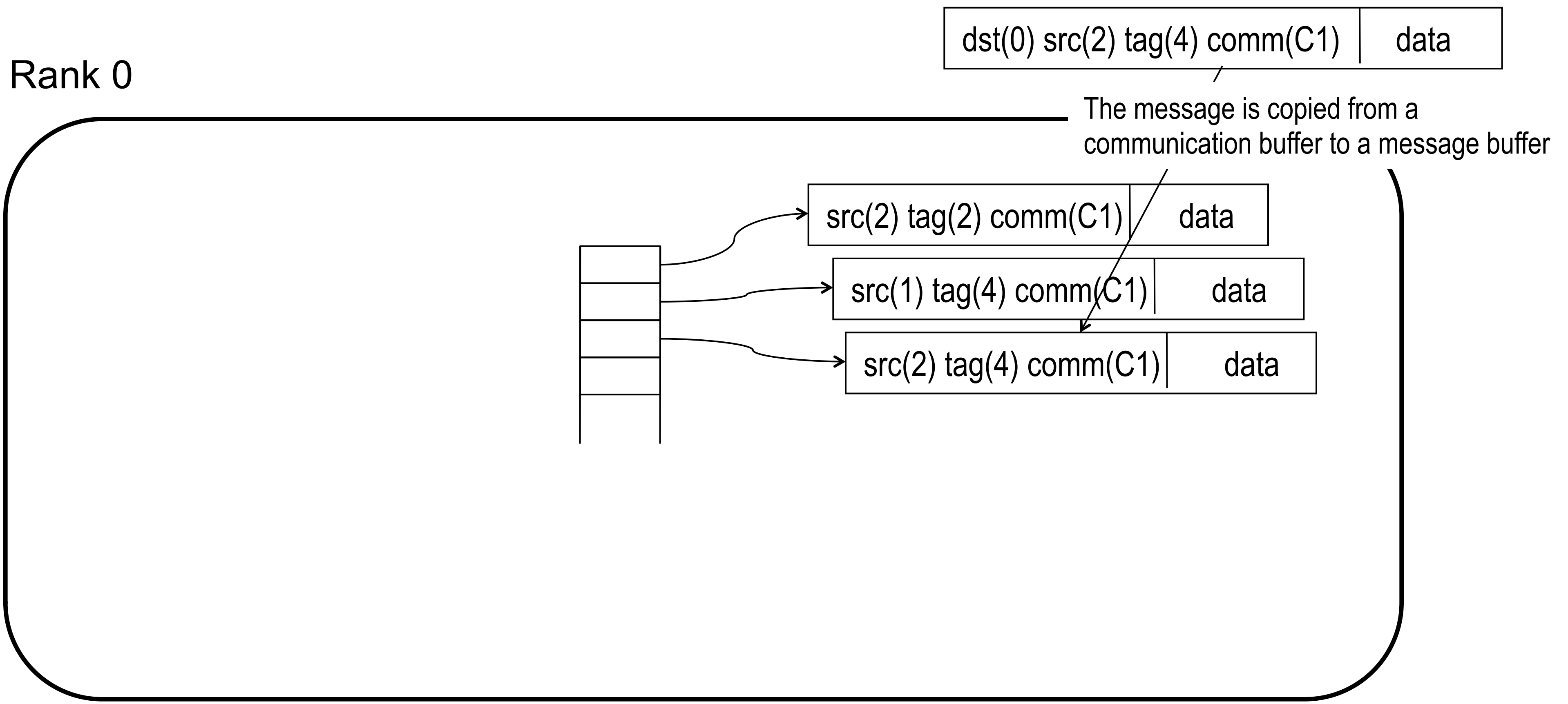
Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```



Inside MPI: Basic

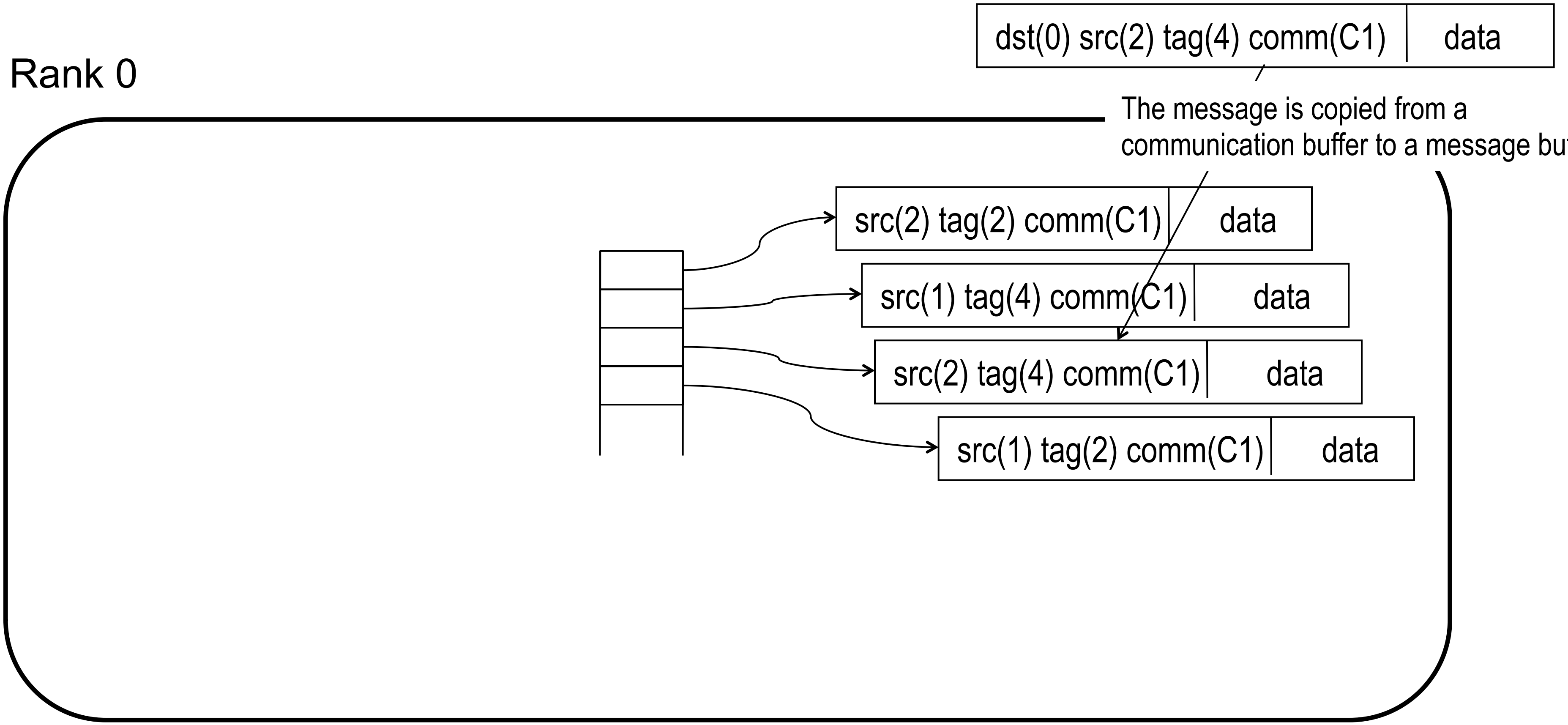
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

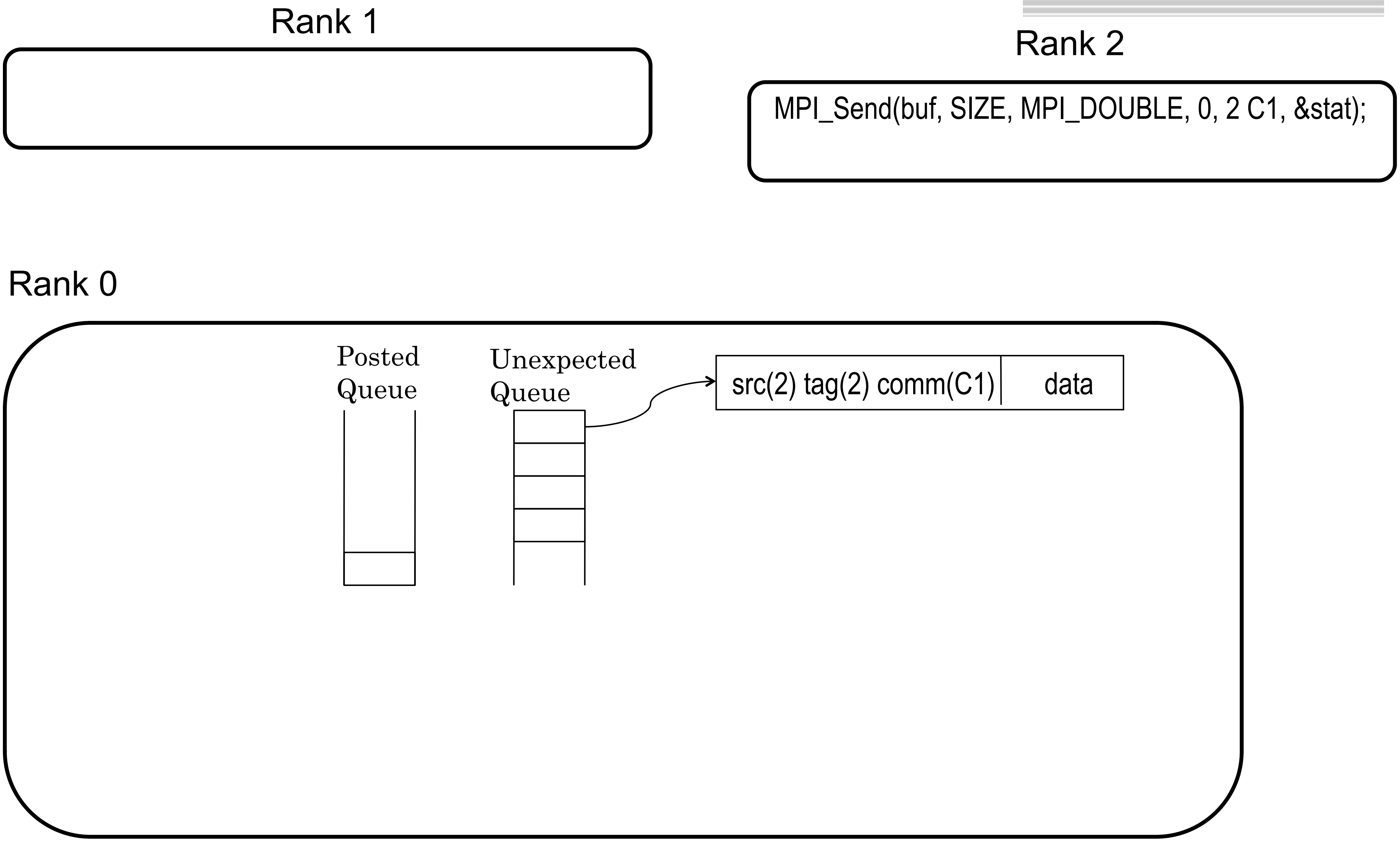
```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0

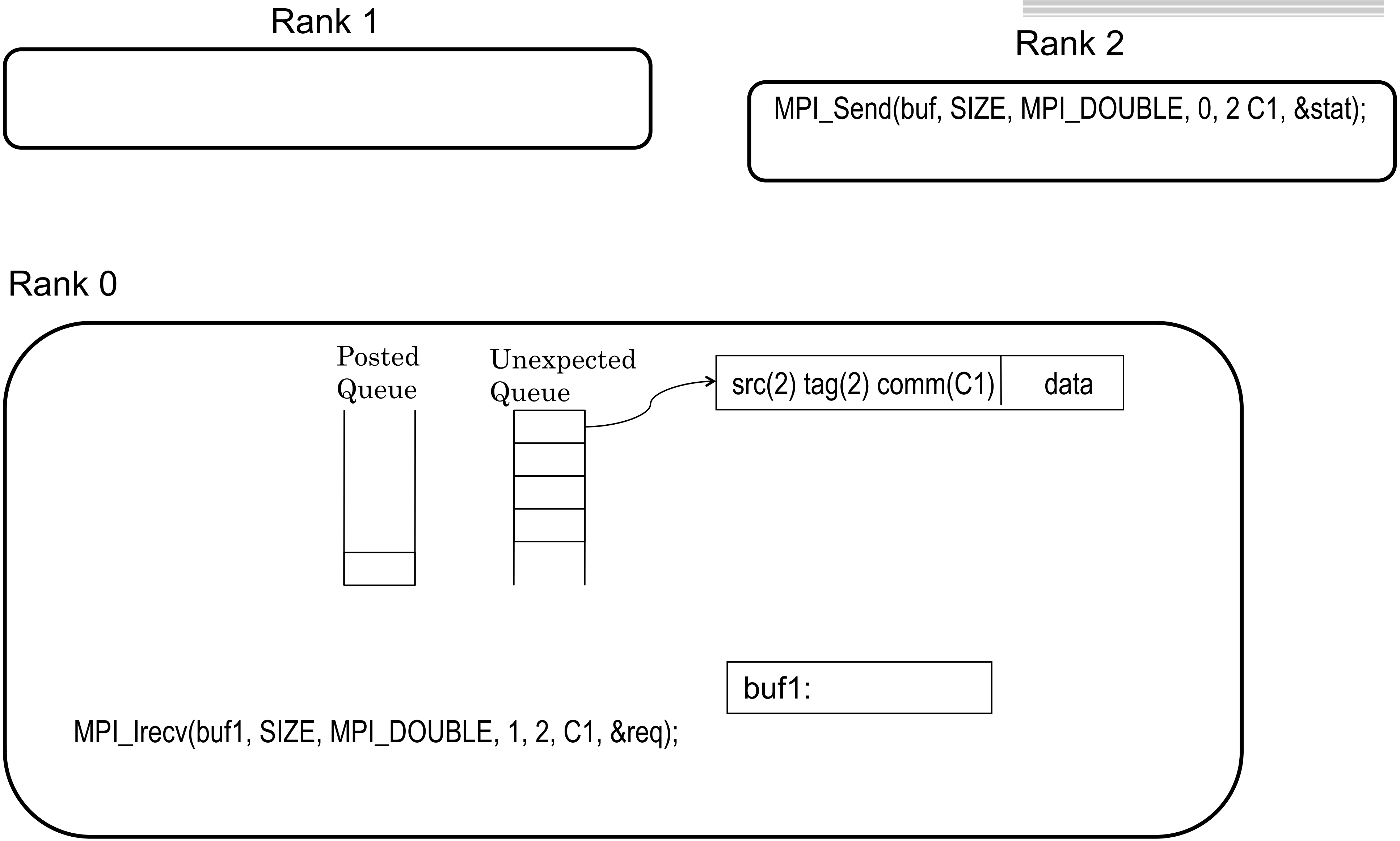


13年8月8日木曜日

Inside MPI: Basic

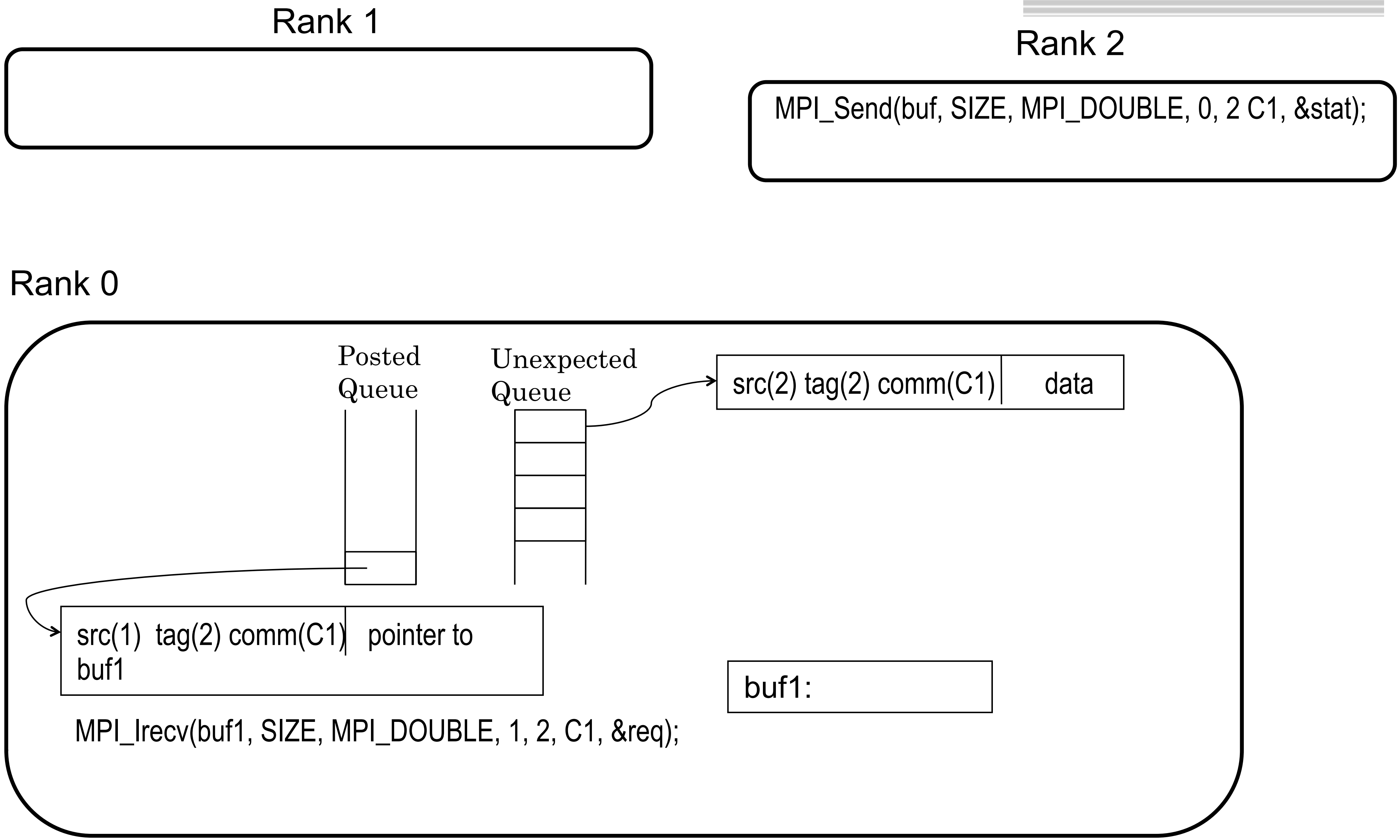


Inside MPI: Basic

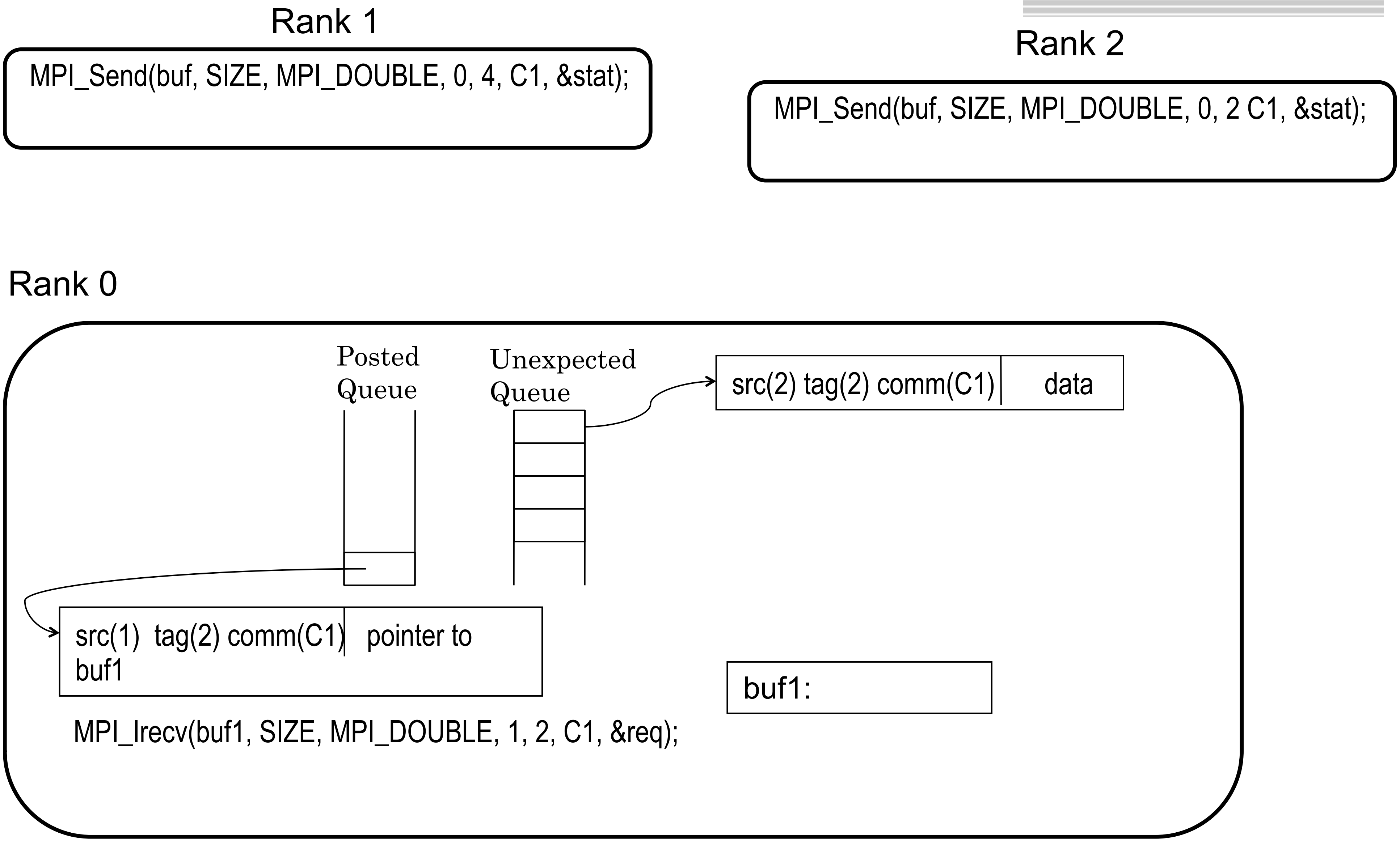


13年8月8日木曜日

Inside MPI: Basic



Inside MPI: Basic



Inside MPI: Basic

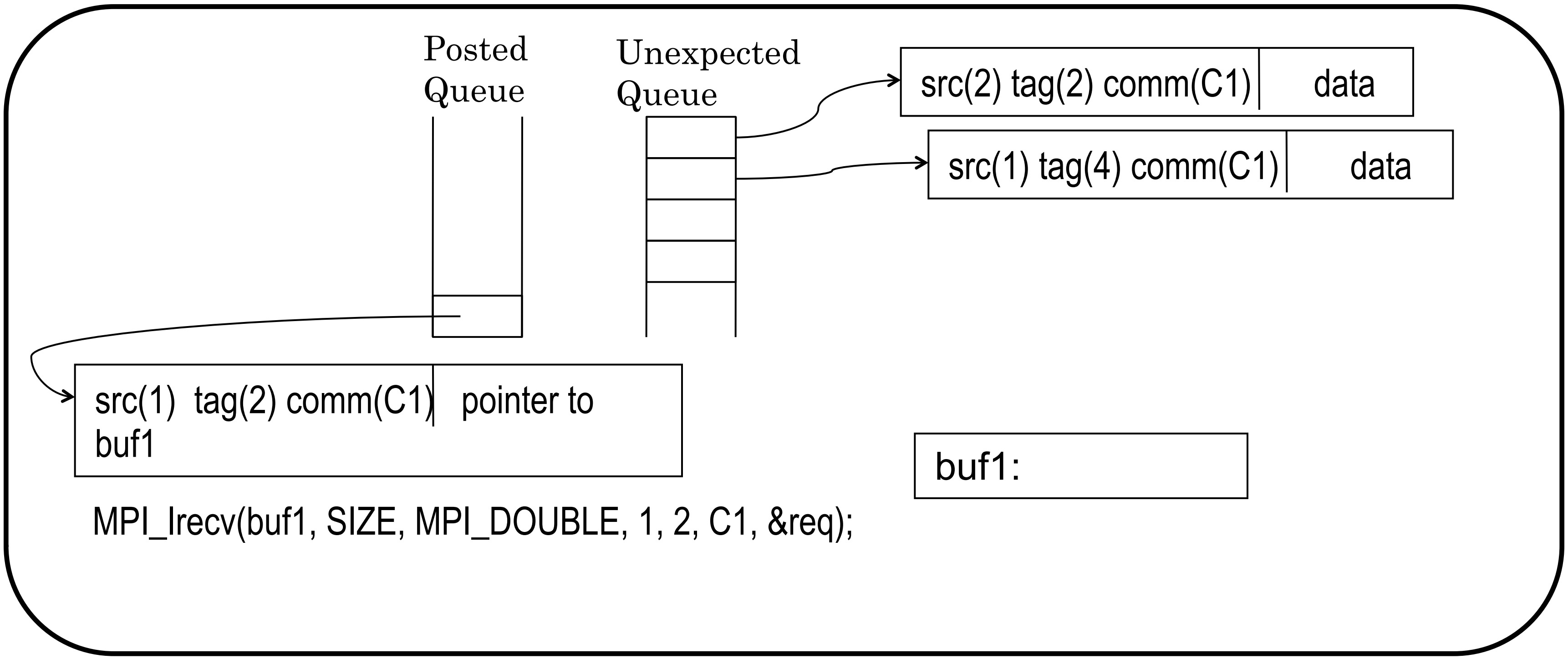
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
```

Rank 0



Inside MPI: Basic

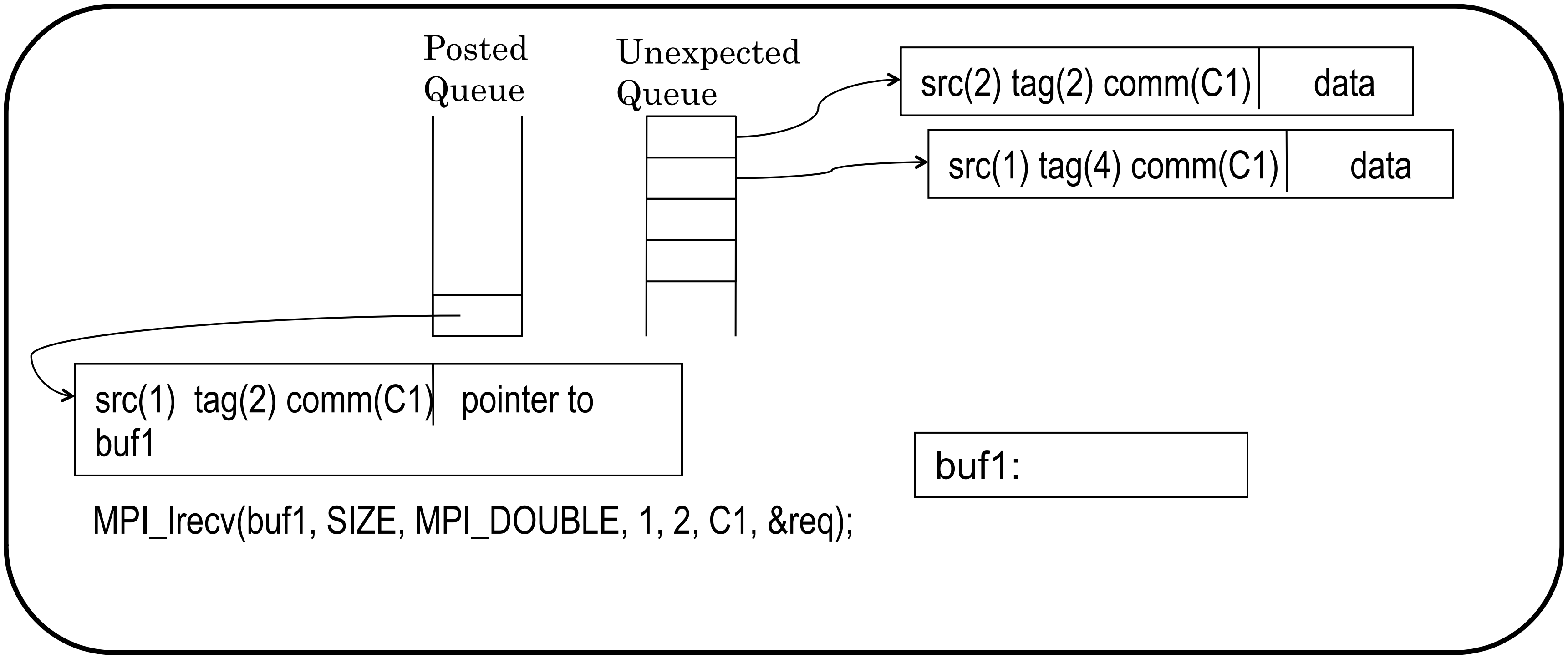
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic

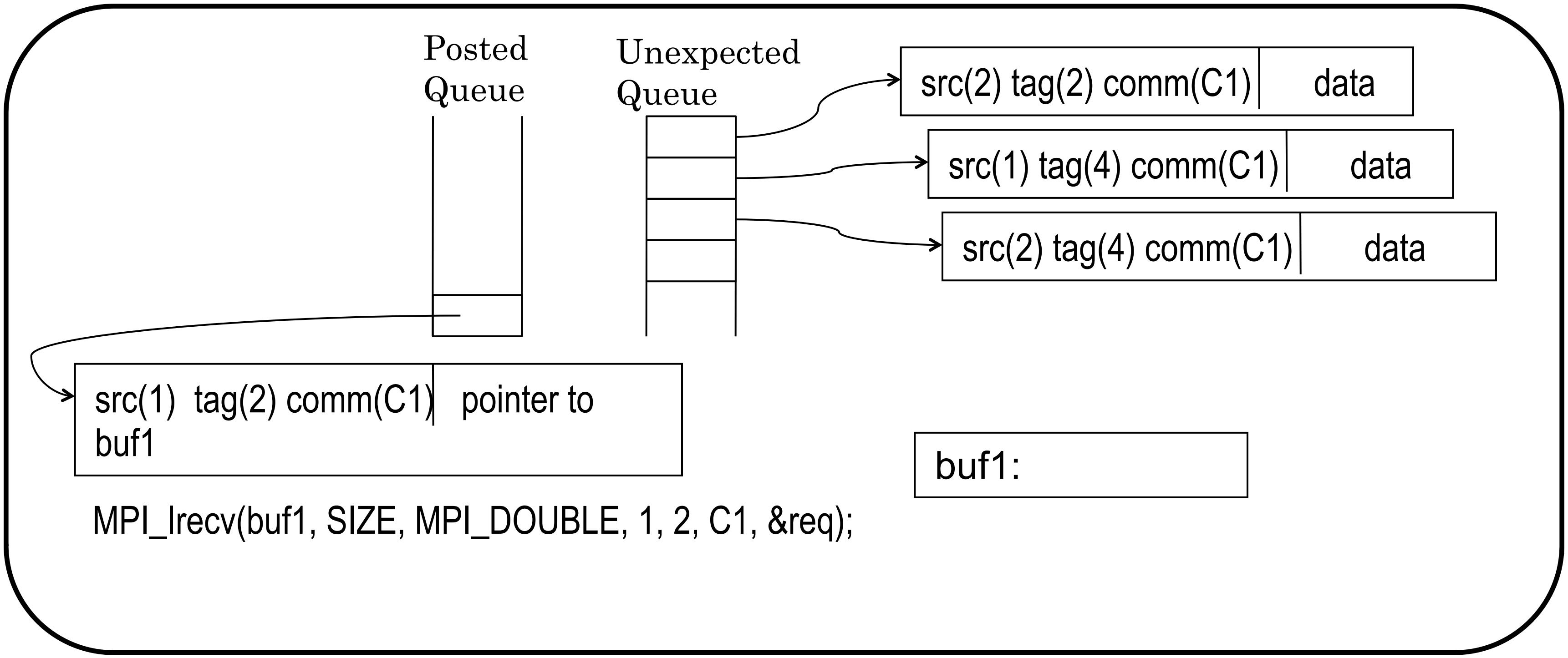
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic

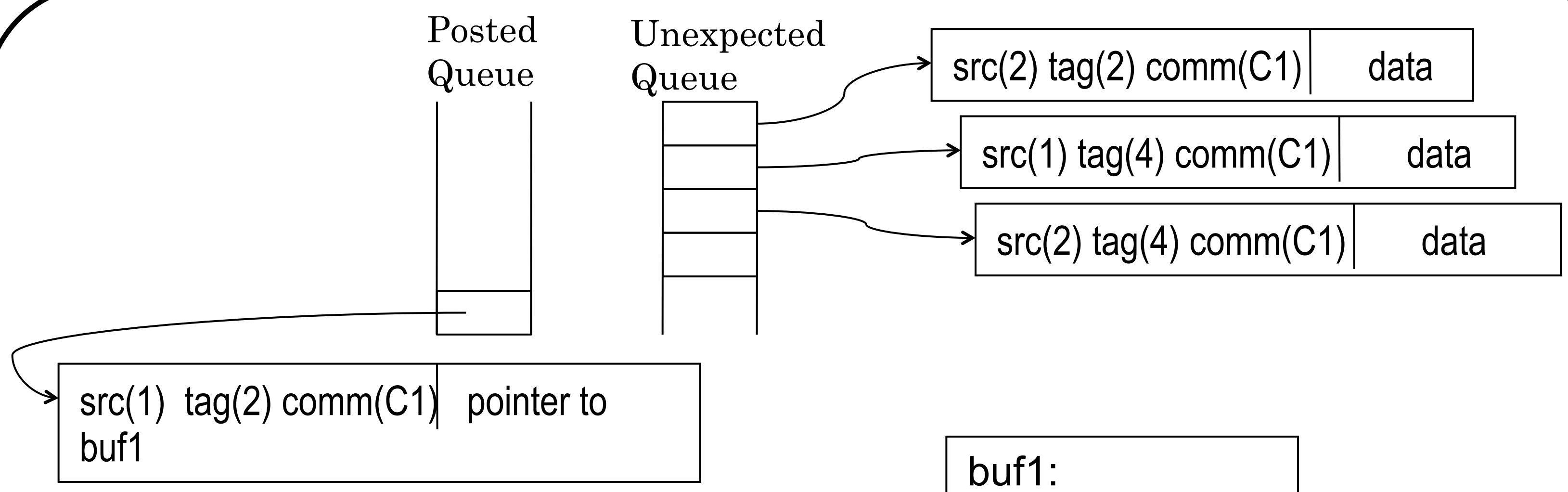
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0




```
MPI_Irecv(buf1, SIZE, MPI_DOUBLE, 1, 2, C1, &req);
```

Inside MPI: Basic

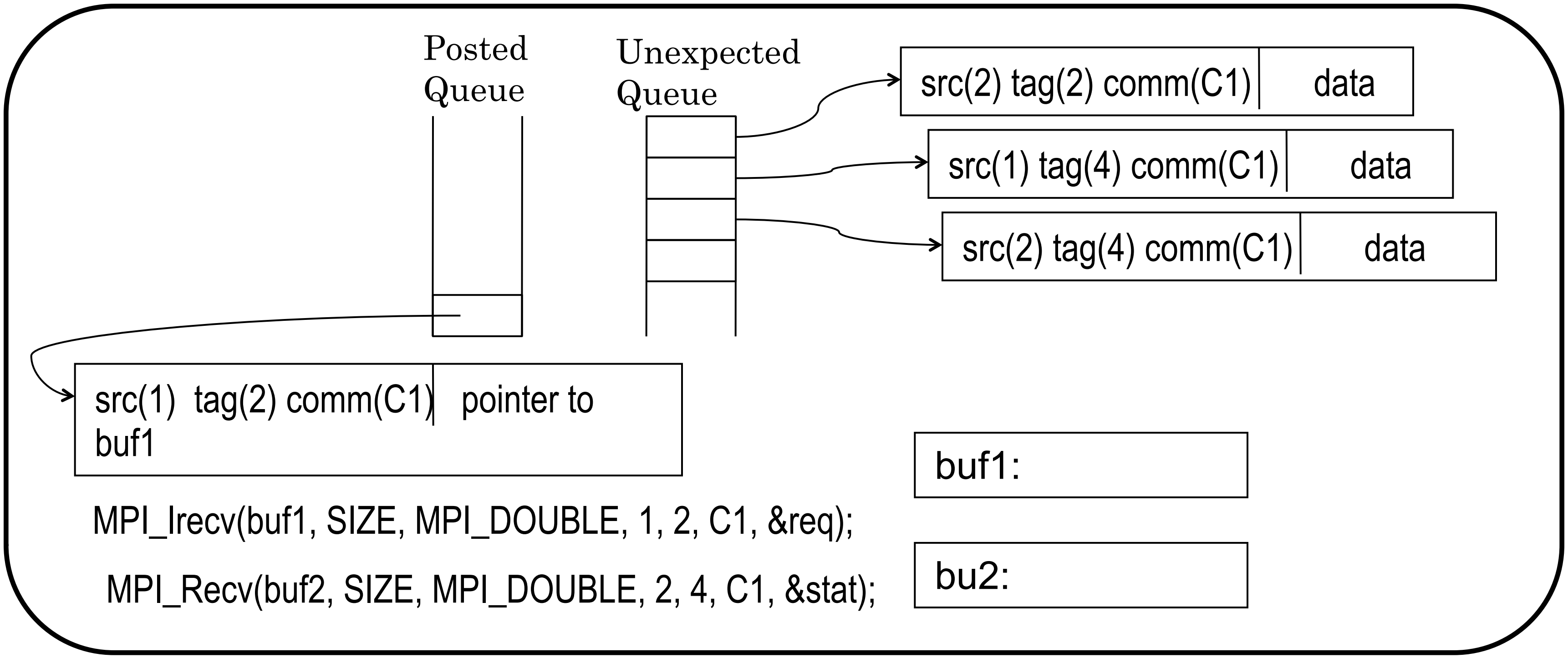
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic

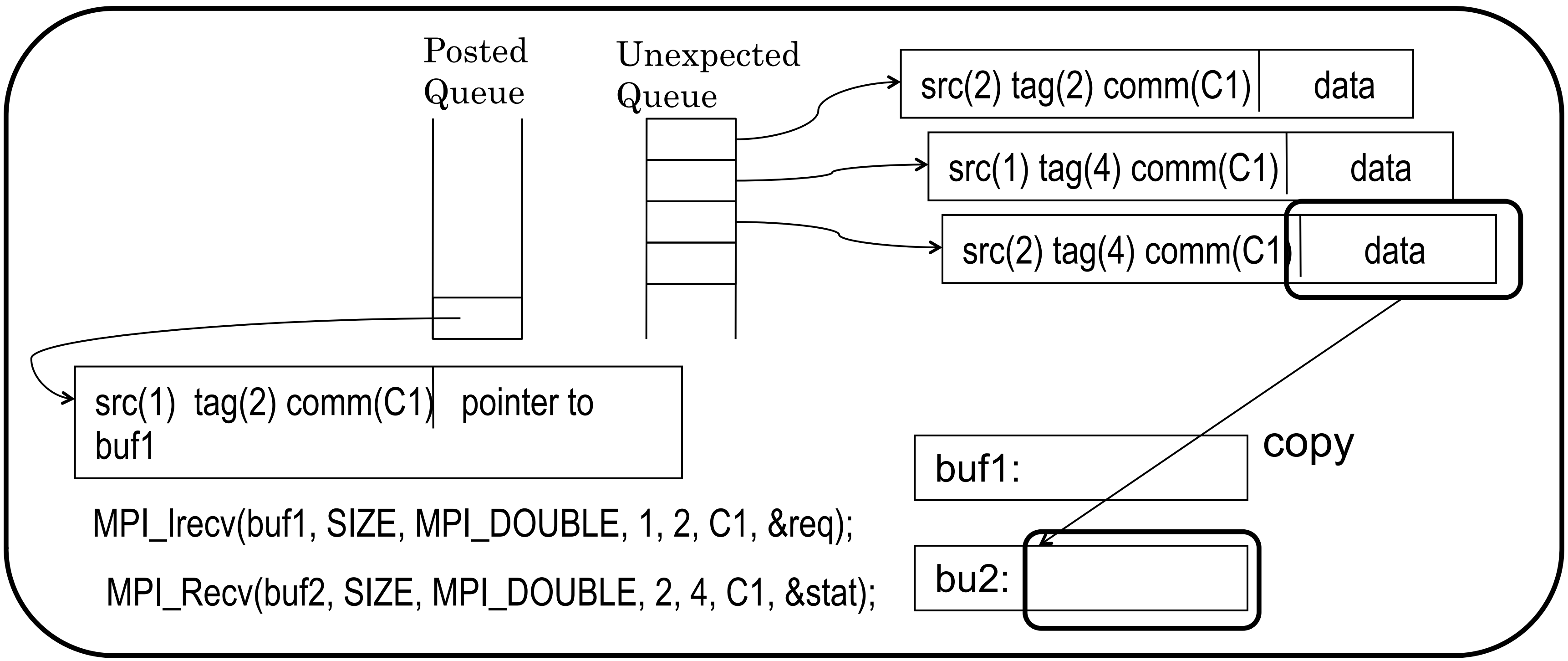
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0



13年8月8日木曜日

Inside MPI: Basic

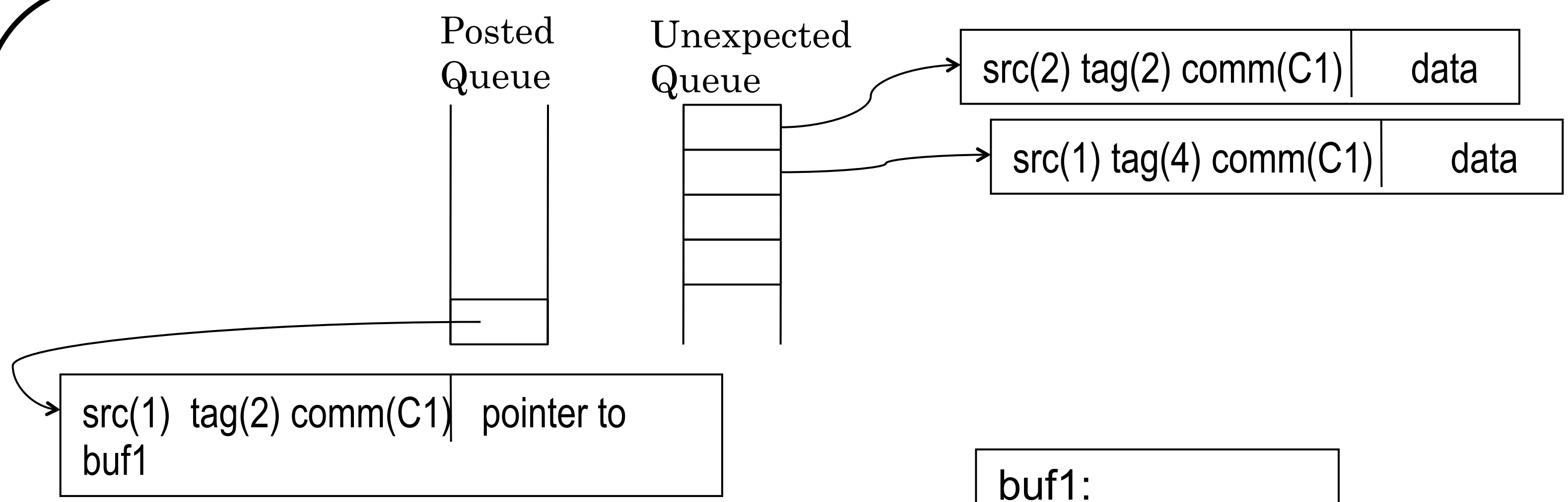
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0




```
MPI_Irecv(buf1, SIZE, MPI_DOUBLE, 1, 2, C1, &req);
MPI_Recv(buf2, SIZE, MPI_DOUBLE, 2, 4, C1, &stat);
```

buf1:

buf2:

Inside MPI: Basic

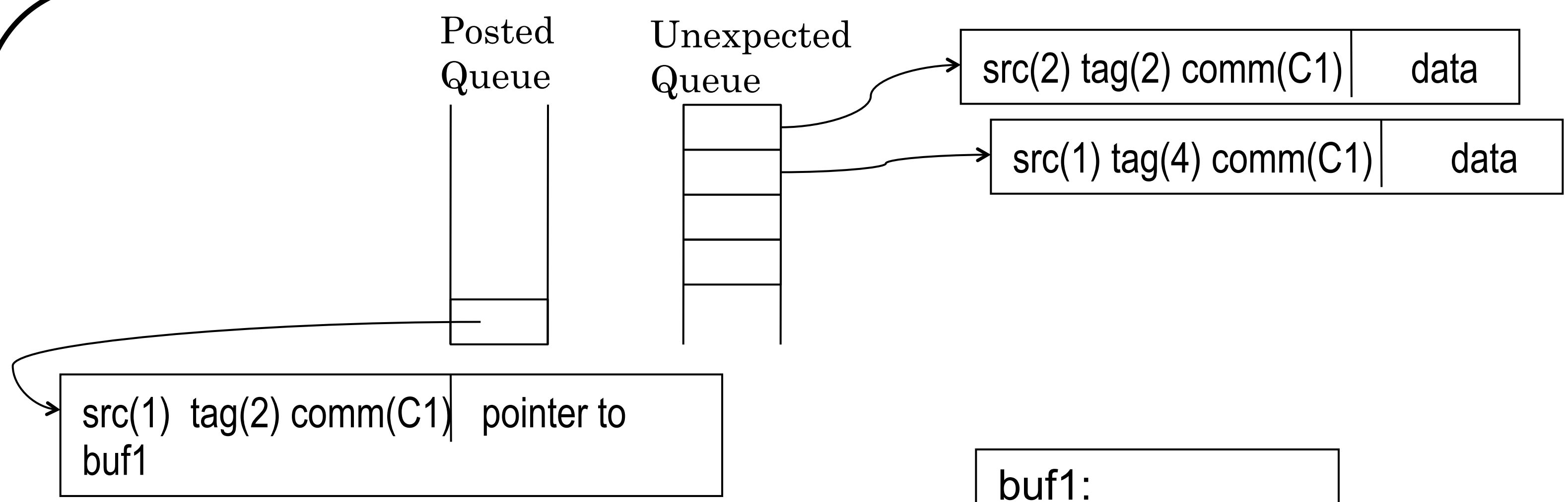
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

Rank 0




```
MPI_Irecv(buf1, SIZE, MPI_DOUBLE, 1, 2, C1, &req);
MPI_Recv(buf2, SIZE, MPI_DOUBLE, 2, 4, C1, &stat);
```

buf1:

buf2:

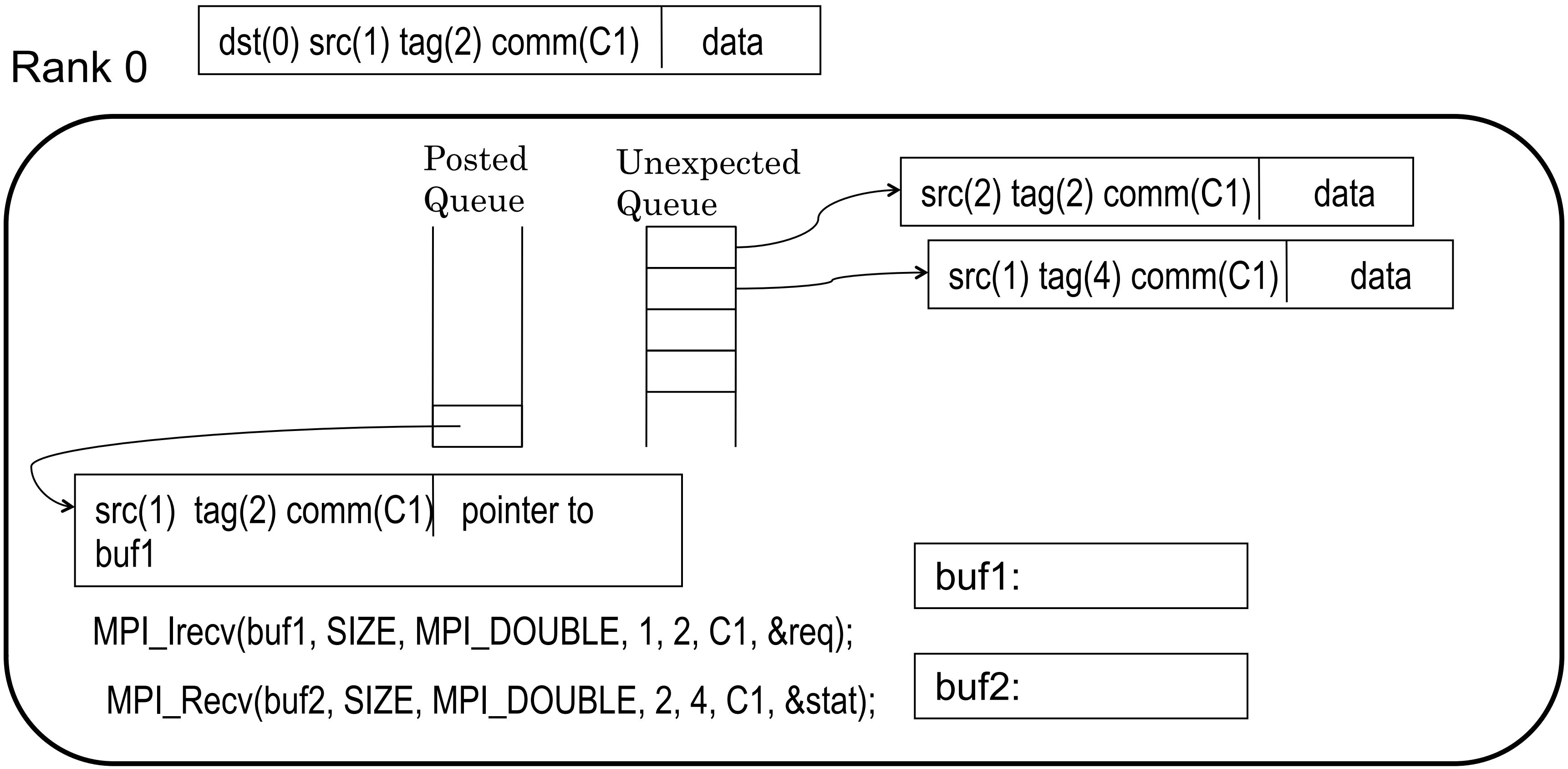
Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```



13年8月8日木曜日

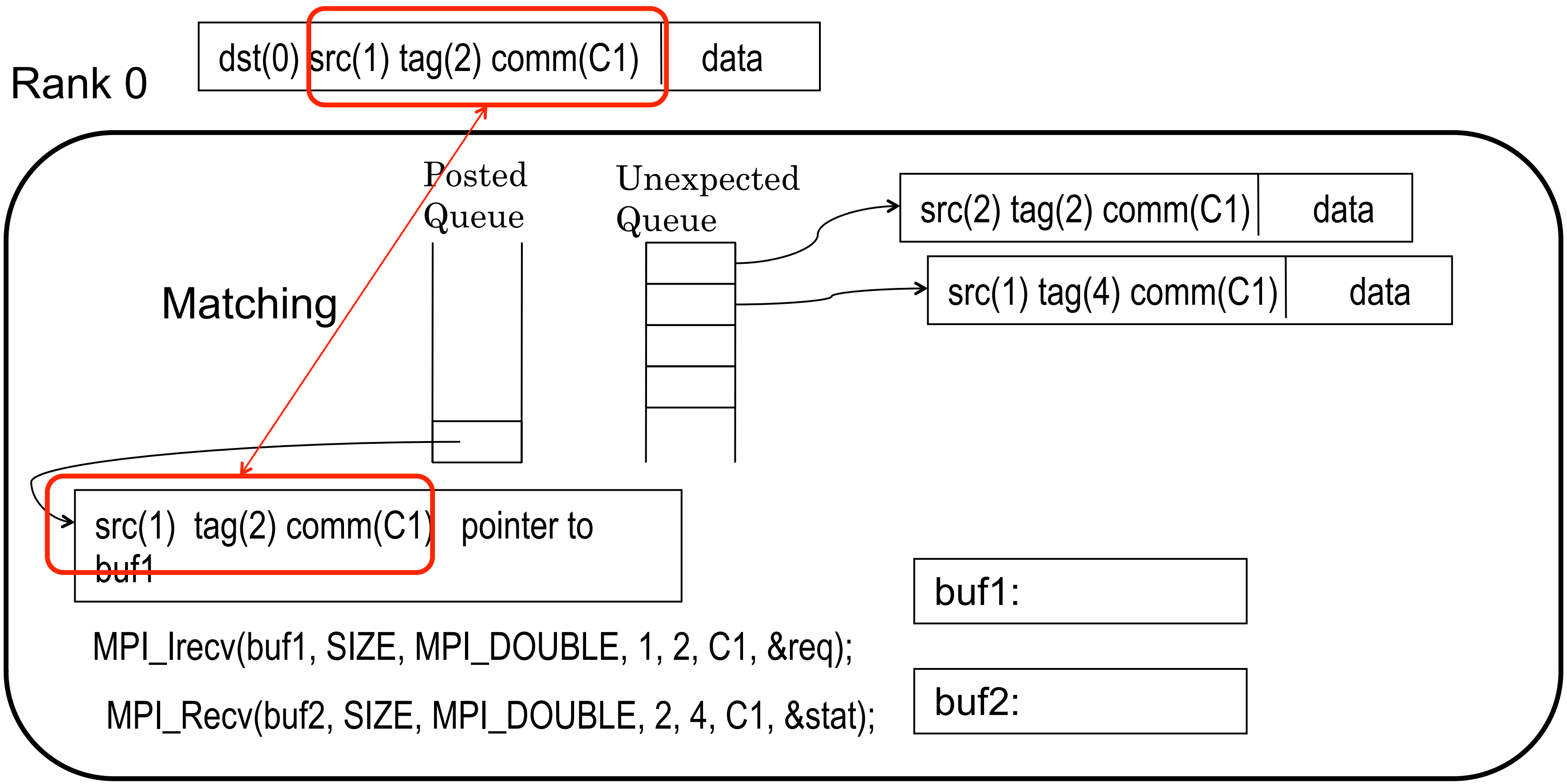
Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```



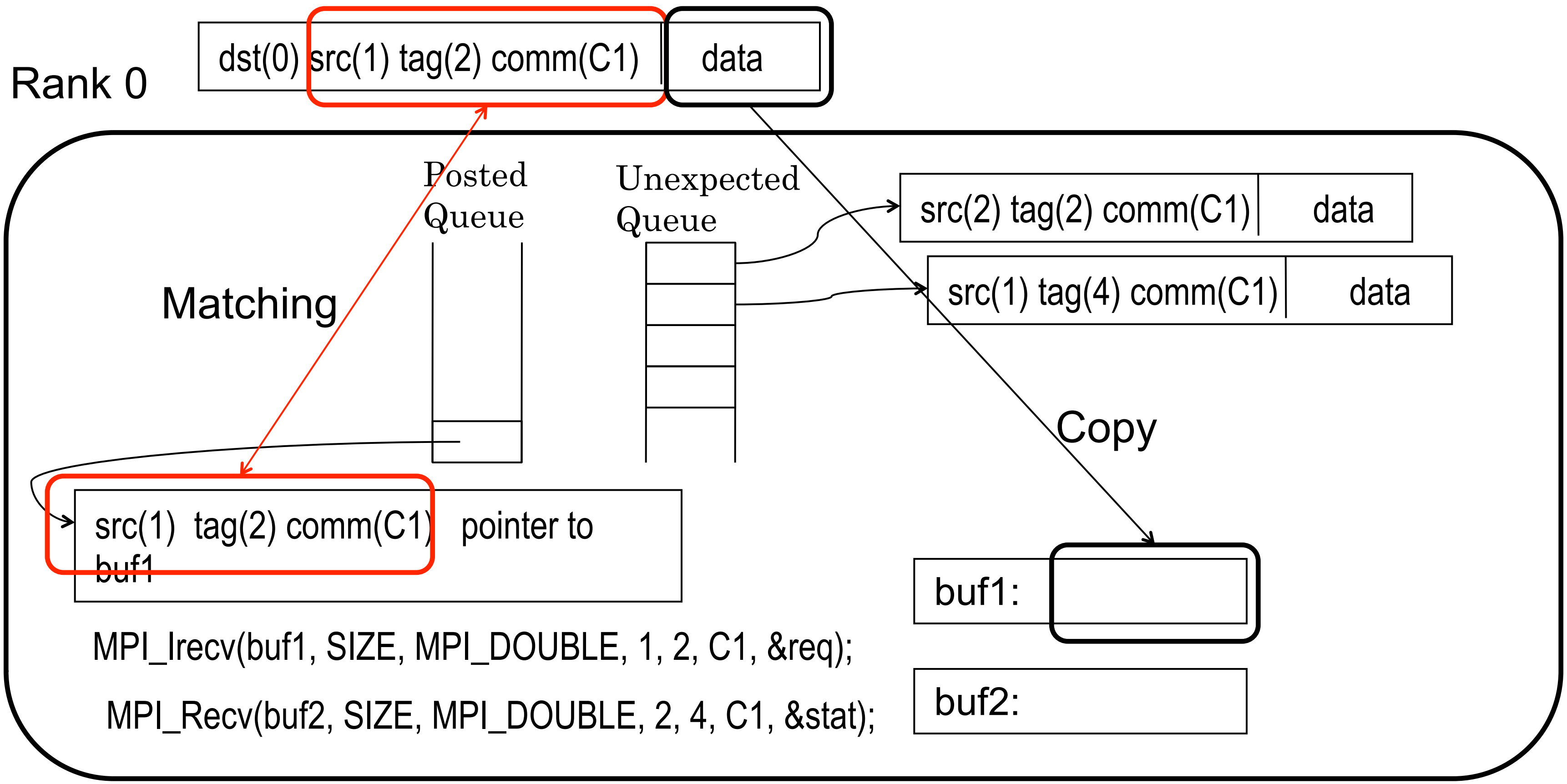
Inside MPI: Basic

Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```



Inside MPI: Basic

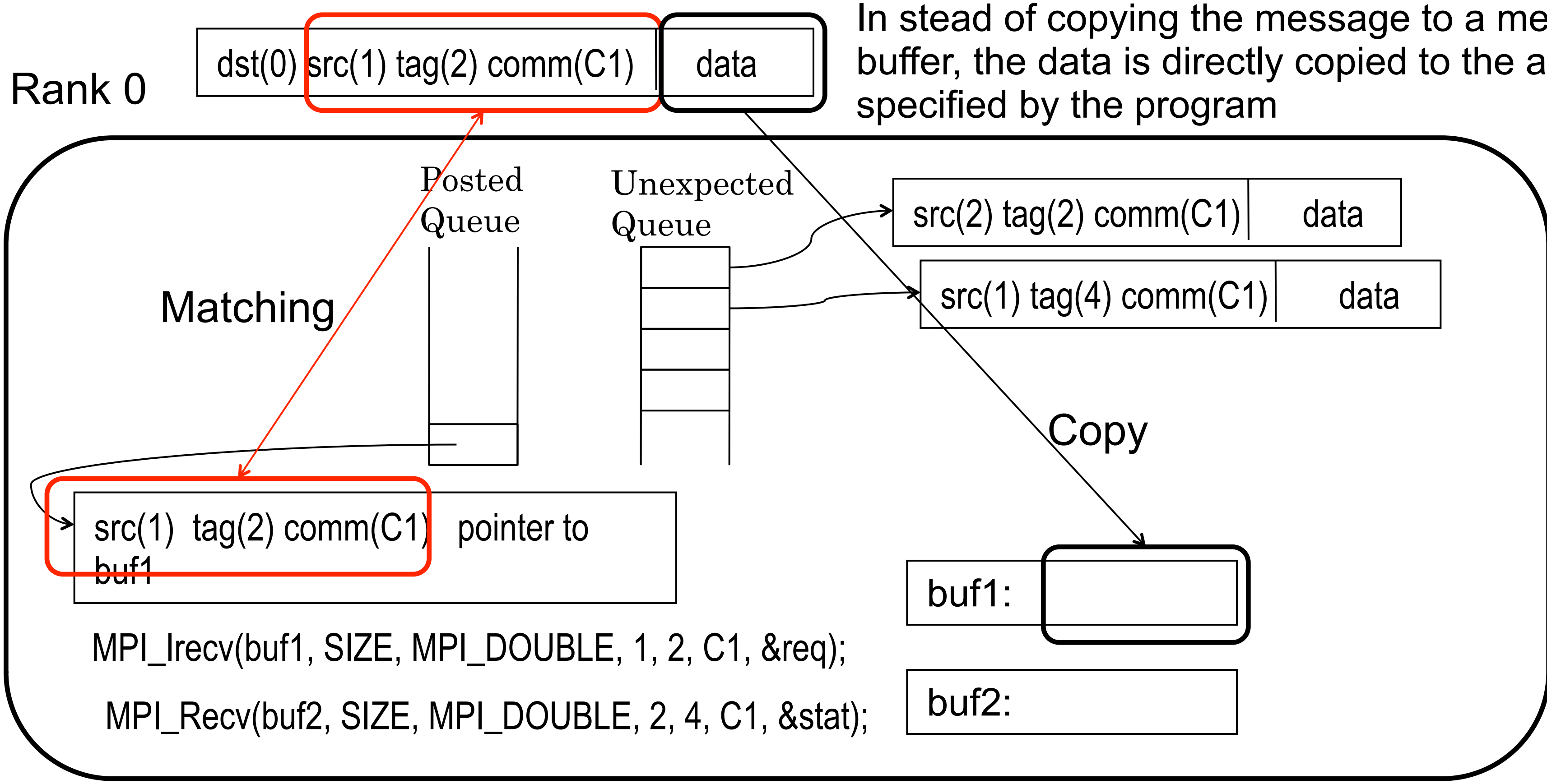
Rank 1

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2, C1, &stat);
```

Rank 2

```
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 2 C1, &stat);
MPI_Send(buf, SIZE, MPI_DOUBLE, 0, 4, C1, &stat);
```

In stead of copying the message to a message buffer, the data is directly copied to the area specified by the program

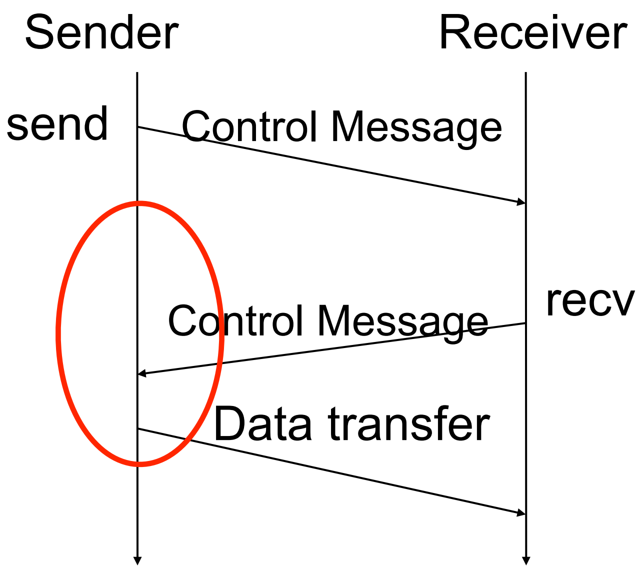


Inside MPI: Basic

- Eager Protocol
  - When a message send primitive is posted, the message is immediately sent to the receiver
  - If MPI_Irecv or MPI_Recv has been posted before the corresponding message arrives, the message may be directly copied to the area specified at the receive primitive.
  - If no receive primitive has been posted, the message is copied to a buffer managed by the MPI library. This means
    - An extra copy overhead from the buffer to the area specified at the receive primitive
    - If the message size is too large, the memory area for buffering cannot be allocated
- MPI Implementation employs another communication protocol called Rendezvous

Inside MPI: Rendezvous

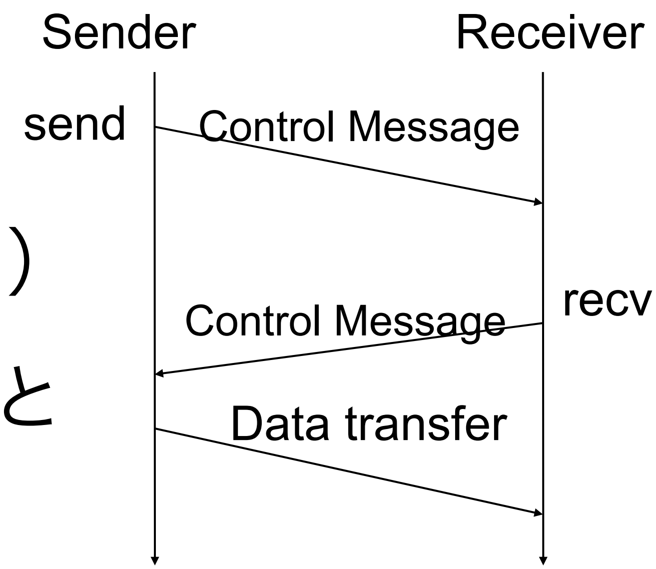
- Rendezvous Protocol
  - When a send primitive is issued, a control message is sent to the receiver.
  - When a receive primitive is issued, another control message is sent to the sender.
  - The sender sends data to the receiver.
  - RDMA (Remote DMA)
    - Using the rendezvous protocol, the sender may transfer data to the receiver using the RDMA facility if the network interface has the RDMA capability
    - RDMA needs the receiver's physical memory address
- The rendezvous protocol involves extra messages to synchronize the sender with the receiver. Adding latency, but data transfer throughput is faster than the eager protocol



If the receiver always specifies the sender rank at a MPI receive primitive, receiver may initiate rendezvous protocol.
Fujitsu implements it and it is called "Hasty Rendezvous".

Rendezvous プロトコルの改良 (Hasty Protocol)

- もし受信側が先だった場合
  - MPI_ANY_SRC でない
  - （かつMPI_ANY_TAG でもない）
  - 受信側が送信側に送信を促すことができる
  - この方が早い



Tips

- The most MPI implementations provide both eager and rendezvous protocols. The protocol is changed according to the message size and can be specified by the environment variable.
  - MPICH2: MPIDI_CH3_EAGER_MAX_MSG_SIZE
  - MVAPICH: MV2_IBA_EAGER_THRESHHOLD, MV2_VBUF_TOTAL_SIZE
  - OpenMPI: I_MPI_EAGER_THRESHOLD (256KB in default)
  - Fujitsu MPI: btl_tofu_eager_limit (13312 ＋ ホップ数 × 296 in default)
    - mpiexec -mca btl_tofu_eager_limit 128000 ./a.out

    or

    - export OMPI_MCA_btl_tofu_eager_limit=128000
- The receive primitive, MPI_Irecv, should be posted as soon as possible in order to get a chance to
  - eliminate extra data copy in the eager protocol
  - reduce the communication latency in the rendezvous protocol

Polling vs. Blocking (or Interrupting)

- When a message has not arrived at the MPI_Recv/ MPI_Wait function, there are two types of waiting methods: Polling and Waiting (or Interrupt)
- Polling
  - MPI library continues to check whether or not a new message arrives until the message arrives
  - It achieves low latency, but
  - It consumes the CPU resource.
  - That is, though the programmer thought the thread might wait for a message and some thread would have a chance to run, it could not run immediately
- Blocking (or Interrupting)
  - MPI library waits for a new message if no messages arrive
  - When a new message arrives, the network device generates an interrupt signal to the CPU. At the interrupt, the device driver is invoked and eventually the execution of MPI library is resume.
  - NO CPU consumption is needed during MPI library wait, but it has higher latency than the polling method

Thread

```
.
MPI_Irecv(…)
.
.
phread_cond_signal(..);
MPI_Wait(…);
.
.
```

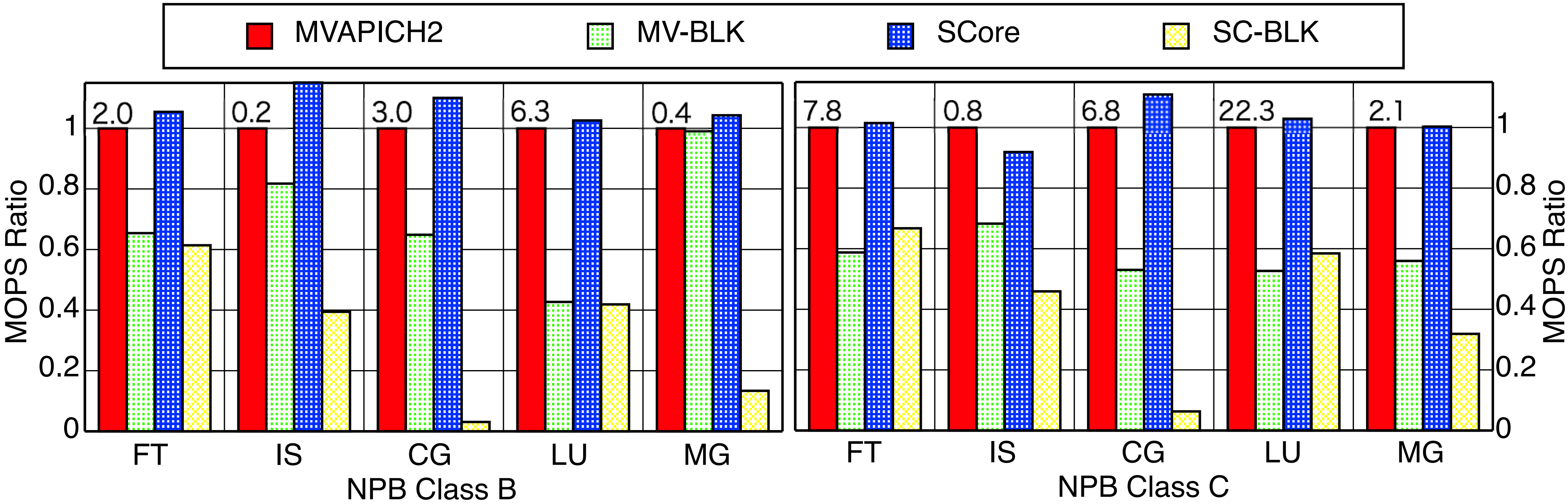
Thread

```
pthread_cond_wait(…);
.

.
```

e.g., OpenMPI has two modes: *aggressive* and *degraded.* If MPI processes more than cores run, the communication progress is under the degraded mode, it may yield the core to others.

e.g., polling is default in MVAPICH2 MV2_USE_BLOCKING=1 enables the blocking mode.

MPI in Blocking Mode



Each node has two Nehalem processors (4 cores, 2.67 GHz). 16 nodes are connected with the QDR Infiniband network. In this evaluation, number of processes is set to 64 (16x4).

Fujitsu MPI Tips

詳細はマニュアルを参照のこと

- MPI_Irecv通信高速化の可能性：Hasty rendezvous protocol
  - mpiexec –mca pml_ob1_use_hasty_rendezvous 1 ./a.out
- Point-to-point通信高速化の可能性
  - mpiexec –mca common_tofu_max_tnis 4 ./a.out

- MPI statistics
  - mpiexec -mca mpi_print_stats 1 ./a.out
  - mpiexec –mca mpi_print_stats 2 –mca mpi_print_stats_ranks 1,4,8 ./a.out
- MPI_Isendにおける送信完了前送信バッファ変更検査
  - mpiexec –mca mpi_check_buffer_write 1 ./a.out

MCAパラメーターmpi_print_statsに値1を指定した場合のMPI統計情報の出力例

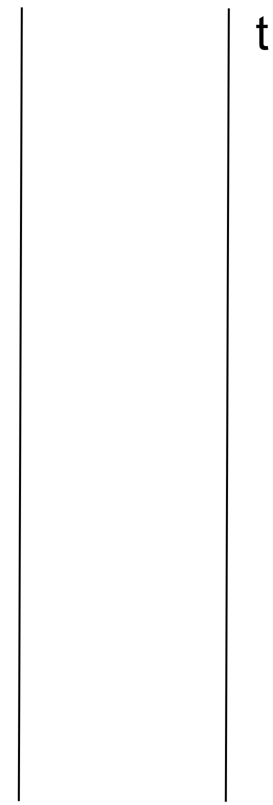
MCAパラメーターmpi_print_statsに値1を指定した場合のMPI統計情報の出力例

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


------------------------- MPI Information -------------------------
Dimension                        3
Shape                        2x3x4

---------------------- MPI Memory Usage (MiB) ----------------------
                              MAX               MIN              AVE
Estimated_Memory_Size       93.90 [    0]     44.39 [    1]     47.29

------------------- Per-peer Communication Count -------------------
                              MAX               MIN              AVE
In_Node                      1024 [    0]         0 [    1]     512.0
Neighbor                     3072 [    1]         0 [    8]    1621.3
Not_Neighbor                 3072 [   11]         0 [    0]     938.7
Total_Count                  3072 [    0]      3072 [    0]    3072.0
Connection                     46 [    0]         9 [    4]      11.8
Max_Hop                         4 [    0]         2 [    4]       3.1
Average_Hop                  2.27 [   35]      1.60 [    6]      1.84

----------------- Per-peer Transmission Size (MiB) -----------------
                              MAX               MIN              AVE
In_Node                    256.00 [    0]      0.00 [    1]    128.00
```

Overlapping Communication and Computation

The typical parallel program

```
do {
    /* local computation */
    /* communicaiton */
} while (…);
```



Overlapping Communication and Computation

```
do {
/* local comp.  and nonblocking comm.1/4 */
/* local comp.  and nonblocking comm.2/4 */
/* local comp.  and nonblocking comm.3/4 */
/* local comp.  and nonblocking comm.4/4 */
/` wait for completion */
} while (…);
```

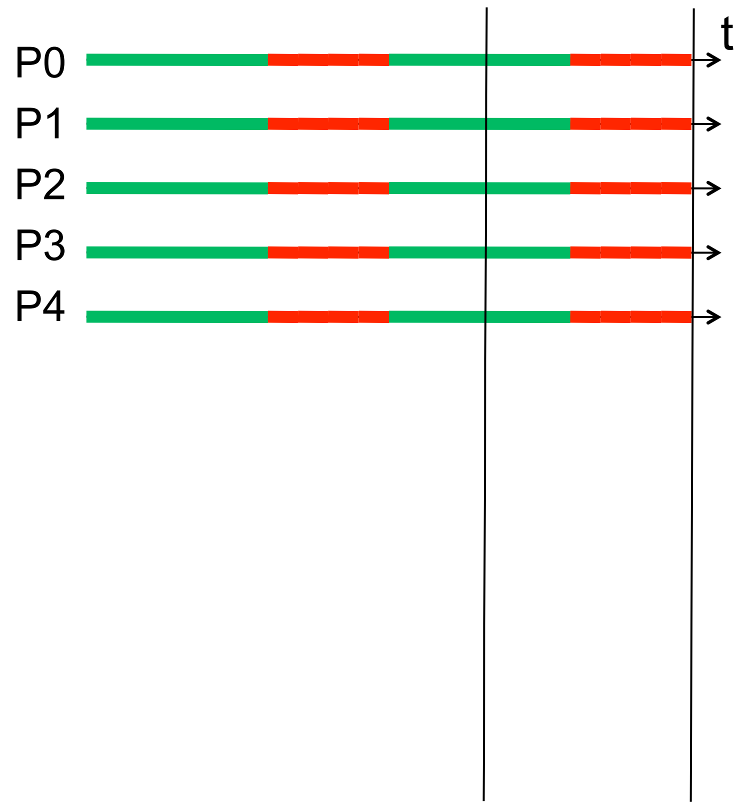
Does MPI really support overlapping communication and computation ?   MPI_Irecv and MPI_Isend

```
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
MPI_Waitall(…);
```

Overlapping Communication and Computation

The typical parallel program

```
do {
    /* local computation */
    /* communicaiton */
} while (…);
```



Overlapping Communication and Computation

```
do {
/* local comp.  and nonblocking comm.1/4 */
/* local comp.  and nonblocking comm.2/4 */
/* local comp.  and nonblocking comm.3/4 */
/* local comp.  and nonblocking comm.4/4 */
/` wait for completion */
} while (…);
```

```
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
MPI_Waitall(…);
```

Does MPI really support overlapping communication and computation ?   MPI_Irecv and MPI_Isend

Overlapping Communication and Computation

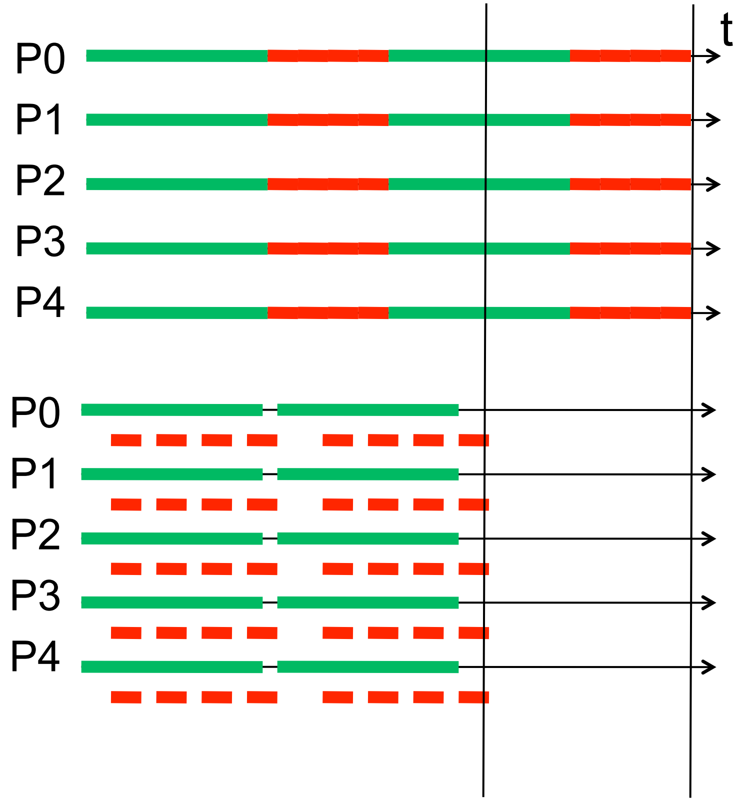
The typical parallel program

```
do {
    /* local computation */
    /* communicaiton */
} while (…);
```

Overlapping Communication and Computation

```
do {
/* local comp.  and nonblocking comm.1/4 */
/* local comp.  and nonblocking comm.2/4 */
/* local comp.  and nonblocking comm.3/4 */
/* local comp.  and nonblocking comm.4/4 */
/` wait for completion */
} while (…);
```

Does MPI really support overlapping communication and computation ?   MPI_Irecv and MPI_Isend




```
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
………;   MPI_Irecv(…); MPI_Isend(…);
MPI_Waitall(…);
```

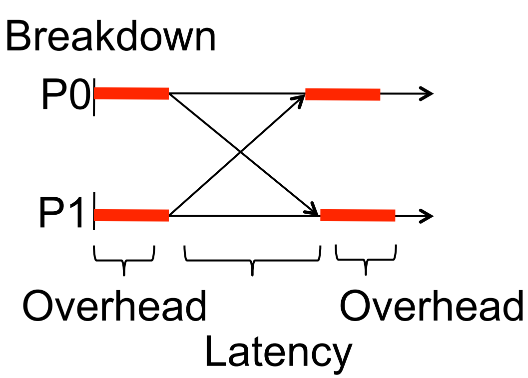

13年8月8日木曜日

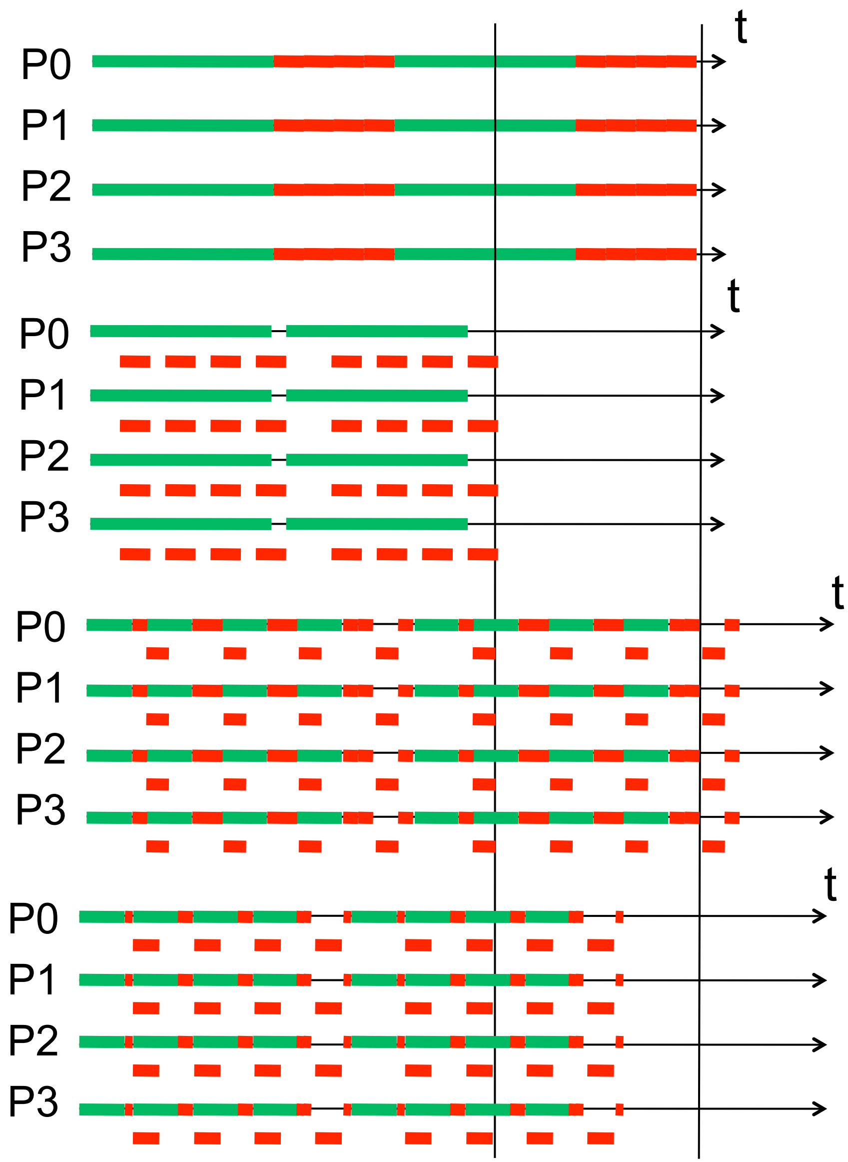
Overlapping Communication and Computation

```
do {
/* local comp.  and nonblocking comm.1/4 */
/* local comp.  and nonblocking comm.2/4 */
/* local comp.  and nonblocking comm.3/4 */
/* local comp.  and nonblocking comm.4/4 */
/` wait for completion */
} while (...);
```

```
do {
/* local comp */ MPI_Irecv(...), MPI_Isend(.....):
/* local comp */ MPI_Irecv(...), MPI_Isend(.....):
/* local comp */ MPI_Irecv(...), MPI_Isend(.....):
/* local comp */ MPI_Irecv(...), MPI_Isend(.....):
MPI_Waitall(...)
} while (...);
```

t





13年8月8日木曜日

Tips

- Overlapping Communication and Computation is a key to carry out strong scaling, but
- Fragmented communications will incur additional overhead which depends on communication library implementation
- Computer Scientists have been studying on better language constructs or facilities with overlapping communication and computation

Other Tips: Persistent Communication

```
MPI_Recv_init((buf, count, MPI_DOUBLE, src, tag,
                    MPI_COMM_WORLD, &req[1]);
MPI_Send_Init(buf, count, MPI_DOUBLE, dest, tag,
                    MPI_COMM_WORKD, &req[0]);
for  (I = 0; …….) {
   /` `/
MPI_Startall(2, req);
MPI_Waitall(2, req, stat);
}
```

```
for (I = 0; …..) {
/* …. */
MPI_Irecv(buf, count, MPI_DOUBLE, src, tag,
                    MPI_COMM_WORKD, &req[0]);
MPI_Isend((buf, count, MPI_DOUBLE, dest, tag,
                    MPI_COMM_WORLD, &req[1]);
MPI_Waitall(2, req, stat);
}
```

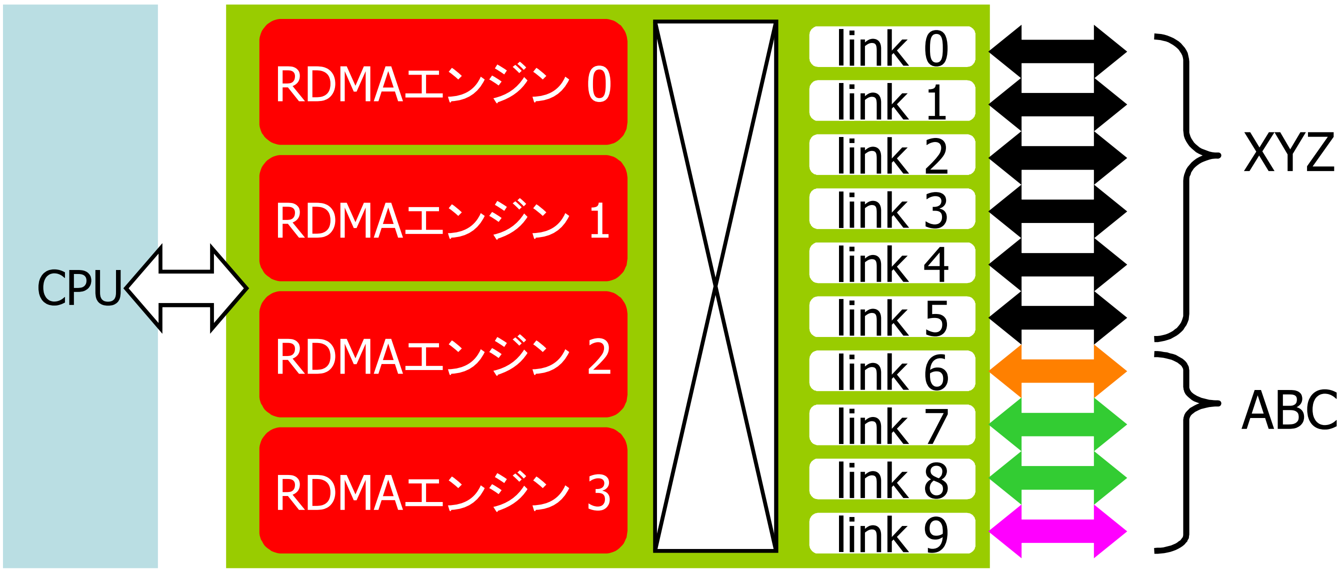
- Whether or not the persistent communication achieves better performance is implementation dependent, but it is much readable (I think).

Persistent Communication (永続通信）の2つの利点

- MPI_XXX_init 時にのみ通信を開始するオーバヘッドが発生する
- 京や BG/Q のように複数のチャネルがある時に有利
  - MPI_Startall に渡された Request の実際の処理の順序は任意（仕様）
  - 複数の送信要求を複数の RDMA にスケジューリングが可能で、有効利用できるため

京の Tofu Network Interface (TNI) - [再掲]

- ポート数10（XYZ軸6ポート＋ABC軸4ポート）
- 4つのRDMAエンジンを搭載、同時に4送信4受信が可能



ノードあたり 理論性能	TSUBAME 2.0 InfiniBand QDR	Cray XE6 Hopper Gemini 1.2	「京」 Tofu Interconnect	IBM Blue Gene/Q 5D-Torus
演算性能	2391 GFlops	153.6 GFlops	128 GFlops	204.8 GFlops
リンク帯域（片方向）	4 GB/s	5.8 GB/s	5 GB/s	2 GB/s
同時通信数	2	1	4	10
同時通信帯域（片方向）	8 GB/s	8.3 GB/s	20 GB/s	20 GB/s

Other Tips: Nonblocking Message

```
void communication() {
....
for (i = 0; i < n; i++) {
    MPI_Isend(sbuf[i], count, MPI_DOUBLE, dst[i], tag, comm, &sreq);
    MPI_Irecv(rbuf[i], count, MPI_DOUBLE, src[i], tag, comm, &rreq[i]);
}
MPI_Waitall(n, rreq, stat);
}
```

- The programmer might think that it is enough to wait for the completion of receiving messages.
- But this program leaks memory
  - When a nonblocking send/receive operation is issued, the MPI runtime reserves an internal structure (object) for data transfer.
  - The object can be only deallocated at MPI_Wait/MPI_Iwait.
- The following program is OK, but the code cannot get the send errors

```
void communication() {
....
for (i = 0; i < n; i++) {
MPI_Isend(sbuf[i], count, MPI_DOUBLE, dst[i], tag, comm, &sreq);
MPI_Request_free(&sreq);
MPI_Irecv(rbuf[i], count, MPI_DOUBLE, src[i], tag, comm, &rreq[i]);
}
MPI_Waitall(n, rreq, stat);
MPI_Barrier();
}
```

Other Tips: Deadlock or not ?

```
void communication() {
....
for (cnt = 1; cnt < SIZE; cnt <<= 1) {
    MPI_Send(sbuf, cnt, MPI_DOUBLE, rank ^ 1, tag, comm);
    MPI_Recv(rbuf, cnt, MPI_DOUBLE, rank ^ 1, tag, comm);
}
}
```

- Whether or not the execution is deadlock depends on the MPI implementation
- Most Implementations
  - Live if the message size is small (using Eager Protocol)
  - Deadlock if the message size is larger (using Rendezvous Protocol)

Other Tips: One-Sided Communication

- One-Sided Communication Primitives in MPI are not true RDMA (Remote DMA)
- Most Implementations use an extra thread for progressing One-Sided Operations
  - The recent MVAPICH implementation does not use an extra thread for basic data type, but still use an extra thread for handling user-defined data type such as vector.
- The assumption, that the user process may utilize all CPU cores, is NOT true