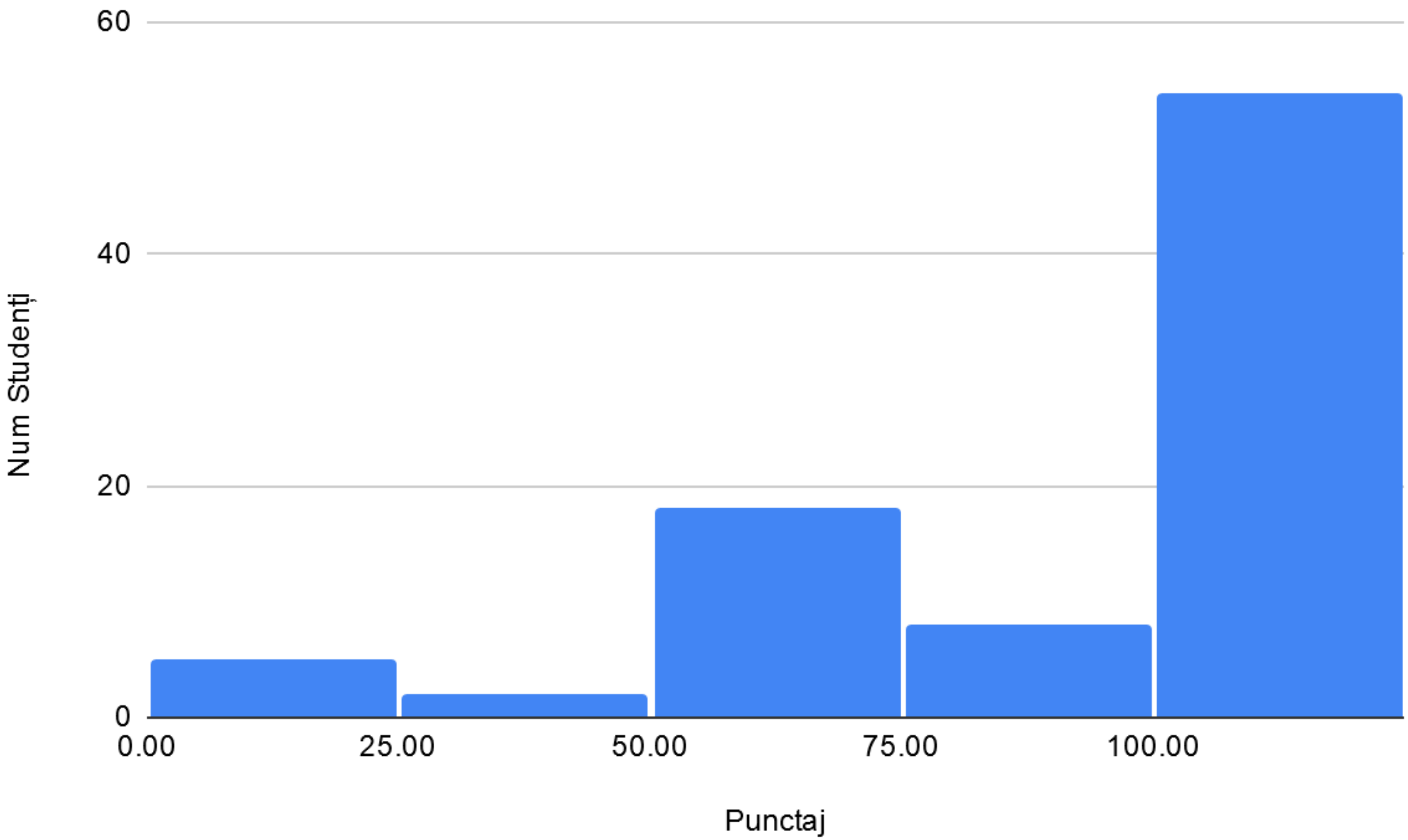
Arhitecturi Paralele
Introducere MPI

Lect. Dr. Ing. Cristian Chilipirea
cristian.chilipirea@mta.ro

Curs susținut în parteneriat cu Prof. Florin Pop

Rezultate Tema 1

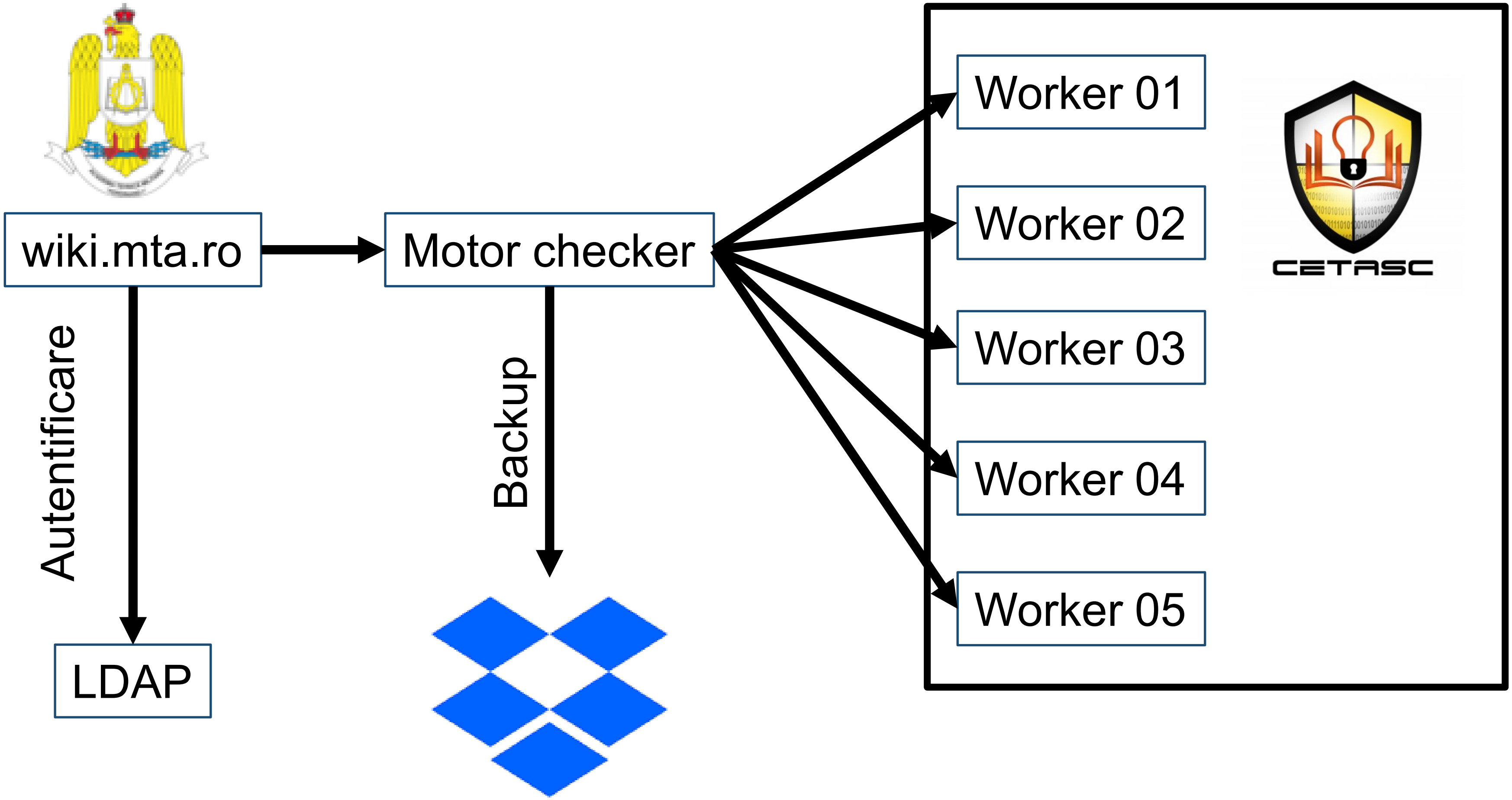


Rezultate Tema 1 - plagiat

- 0 cazuri plagiat
  - **Keep it that way :D**

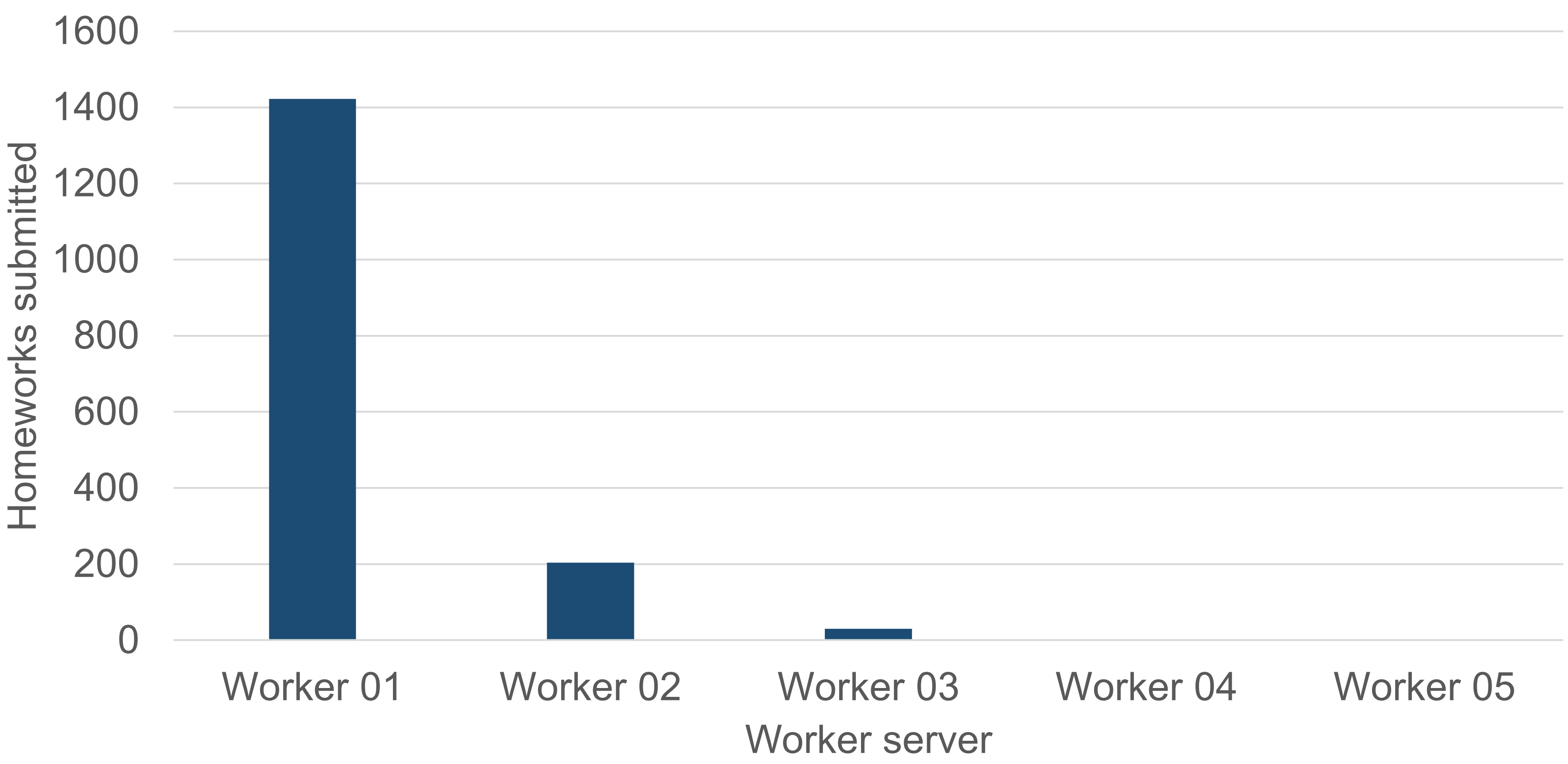
- 8 cazuri în care am fost îngrijrați
  - De minimizat cooperare la temele următoare
  - **Reminder: e vinovat și cine e ajutat și cine ajută**

Checker-ul de teme, arhitectura unui sistem distribuit



Folosire checker tema 01

Între 5 zile si 17 zile folosire continuă a unui sistem quad-core



Programare distribuită

“Studierea unui neuron se numește neuroștiință. Studierea a doi neuroni se numește psihologie.”

În cazul nostru, programarea distribuită reprezintă programarea a cel puțin două sisteme de calcul pentru rezolvarea unei probleme.

MPI

Framework care facilitează

- Pornirea programelor distribuite (procese pe același sistem sau pe sisteme diferite, dar strâns conectate – ideal aceeași rețea)
- Conectarea proceselor unui program distribuit (accept, bind, connect)
- Simplificarea identificării (identificatori în loc de IP, port)
- Simplificarea comunicării (oferă funcții gen Send/Recv, Broadcast)
- Asigură comunicarea corectă pe sisteme cu arhitecturi de calcul diferite (little/big endian problems)

MPI memoria

- Nu avem memorie partajată în MPI. Arhitectură **NUMA**
- Toate variabilele sunt locale proceselor.
- Pentru a muta informație de la un proces la altul vor trebuie folosită comunicație, prin apelul funcțiilor oferite de MPI:
  - Send/Recv
  - Broadcast
  - Scatter
  - Gather

Instalare OpenMPI

```
apt-get install libopenmpi-dev openmpi-bin openmpi-doc openmpi-common
```

Compiling and running MPI programs

mpicc test.c

Pornește 4 **procese**.
Dacă este setat, va porni procesele pe mașini diferite.
Procesele sunt identice dar au id-uri diferite.
Funcționează parțial și cu programe care nu sunt implementate pentru MPI.

mpirun –np **4** a.out
mpirun –np **3** date

./a.out

Funcționează dar pornește **un singur** proces.

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

Pornește procesele MPI

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

Întoarce ID-ul procesului (rank-ul)

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

Întoarce numărul total de procese

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

Afișează hello (pentru fiecare proces pornit).

MPI example

```
#include<mpi.h>
#include<stdio.h>

int main(int argc, char * argv[])
{
        int rank;
        int nProcesses;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
        printf("Hello from %i/%i\n", rank, nProcesses);
        MPI_Finalize();
        return 0;
}
```

Oprește programul MPI.

MPI example executed

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

int main(int argc, char * argv[])
{
	int rank;
	int nProcesses;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
	printf("Hello from %i/%i\n", rank, nProcesses);
	MPI_Finalize();
	return 0;
}
```

Hello from 0/4


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

int main(int argc, char * argv[])
{
	int rank;
	int nProcesses;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
	printf("Hello from %i/%i\n", rank, nProcesses);
	MPI_Finalize();
	return 0;
}
```


Hello from 3/4


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

int main(int argc, char * argv[])
{
	int rank;
	int nProcesses;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
	printf("Hello from %i/%i\n", rank, nProcesses);
	MPI_Finalize();
	return 0;
}
```


Hello from 2/4


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

int main(int argc, char * argv[])
{
	int rank;
	int nProcesses;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &nProcesses);
	printf("Hello from %i/%i\n", rank, nProcesses);
	MPI_Finalize();
	return 0;
}
```


Hello from 1/4

MPI_Send/MPI_Recv

int MPI_Send(↓ void *b, ↓ int c, ↓ MPI_Datatype d, ↓ int reiceiver, ↓ int t, ↓ MPI_Comm)

void *b	int c	MPI_Datatype d	int reiceiver	int t	MPI_Comm
v &v[3] &a v+5	num_el(v) [0,..)	MPI_INT MPI_CHAR MPI_FLOAT MPI_LONG	[0, **num_tasks**)	[0, ..)	MPI_COMM_WORLD

MPI_Send/MPI_Recv

int MPI_Recv(↑ void *b, ↓ int c, ↓ MPI_Datatype d, ↓ int sender, ↓ int t, ↓ MPI_Comm, ↑ MPI_Status *)

v
&v[3]
&a
v+5

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

[0, ..)
MPI_ANY_TAG

[0, **num_tasks**)
MPI_ANY_SOURCE

num_el(v)
[0,..)

MPI_COMM_WORLD

&Stat
MPI_STATUS_IGNORE
Stat.MPI_SOURCE, Stat.MPI_TAG

MPI_Bcast

int MPI_Bcast (↕ void *b, ↓ int c, ↓ MPI_Datatype d, ↓ int root, ↓ MPI_Comm)

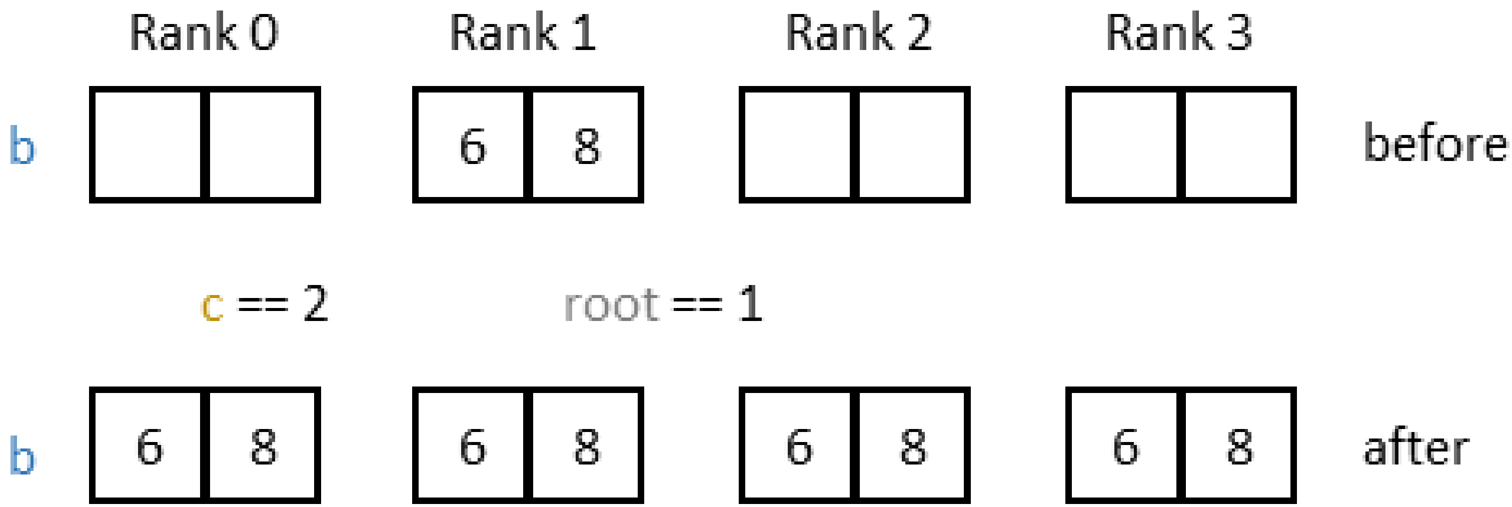
v num_el(v)
&v[3] [0,..)
&a
v+5

[0, **num_tasks**)

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

MPI_COMM_WORLD

MPI_Bcast


Rank 0
Rank 1
Rank 2
Rank 3
b
6
8
before
c == 2
root == 1
b
6
8
6
8
6
8
6
8
after

MPI_Scatter

int MPI_Scatter (↓ void *sb, ↓ int sc, ↓ MPI_Datatype sd, ↑ void *rb, ↓ int rc, ↓ MPI_Datatype rd, ↓ int root, ↓ MPI_Comm)

v
&v[3]
&a
v+5

num_el(v)/**num_tasks**
[0,..)

v
&v[3]
&a
v+5

num_el(v)/**num_tasks**
[0,..)

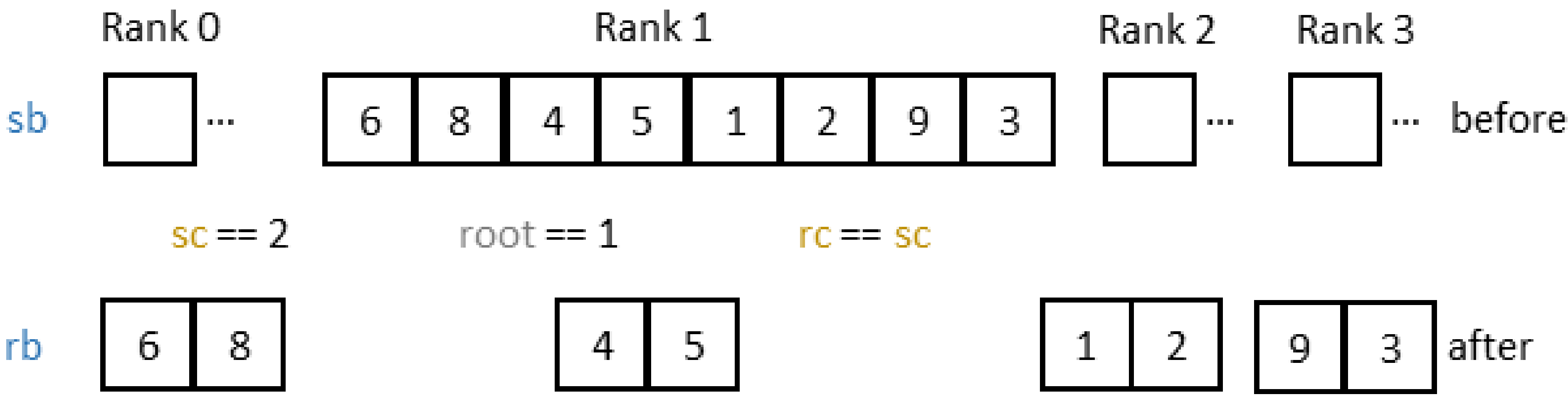
[0, **num_tasks**)

MPI_COMM_WORLD

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

MPI_Scatter



Rank 0
Rank 1
Rank 2
Rank 3
sb
...
6
8
4
5
1
2
9
3
...
...
before
sc == 2
root == 1
rc == sc
rb
6
8
4
5
1
2
9
3
after

MPI_Gather

int MPI_Gather (↓ void *sb, ↓ int sc, ↓ MPI_Datatype sd, ↑ void *rb, ↓ int rc, ↓ MPI_Datatype rd, ↓ int root, ↓ MPI_Comm)

v
&v[3]
&a
v+5

num_el(v)/**num_tasks**
[0,..)

v
&v[3]
&a
v+5

[0, **num_tasks**)

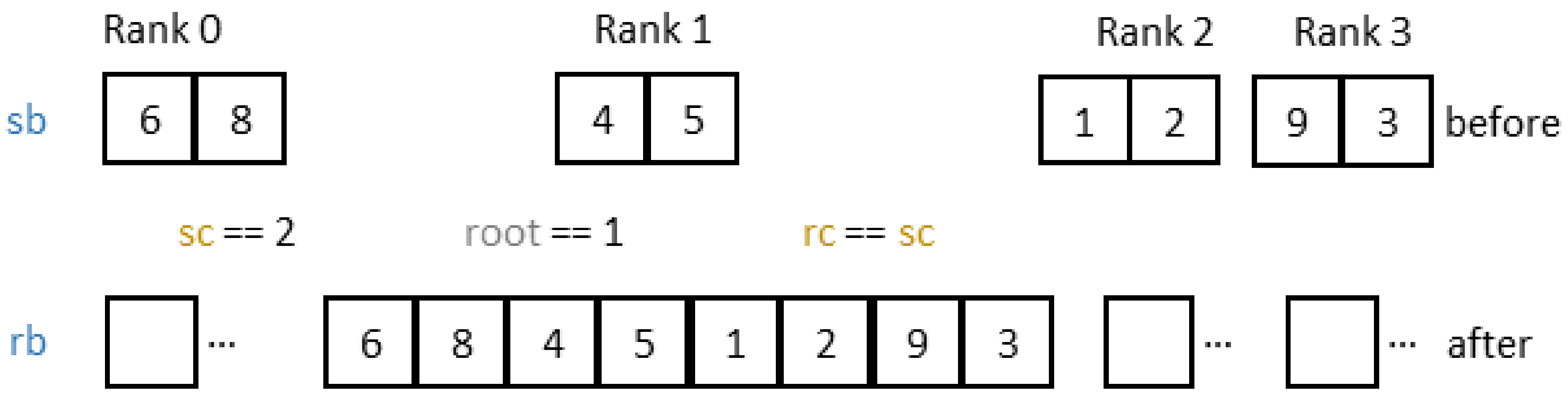
num_el(v)/**num_tasks**
[0,..)

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

MPI_INT
MPI_CHAR
MPI_FLOAT
MPI_LONG

MPI_COMM_WORLD

MPI_Gather


Rank 0
Rank 1
Rank 2
Rank 3
sb
6
8
4
5
1
2
9
3
before
sc == 2
root == 1
rc == sc
rb
...
6
8
4
5
1
2
9
3
...
...
after

MPI blocking/non-blocking send/recv

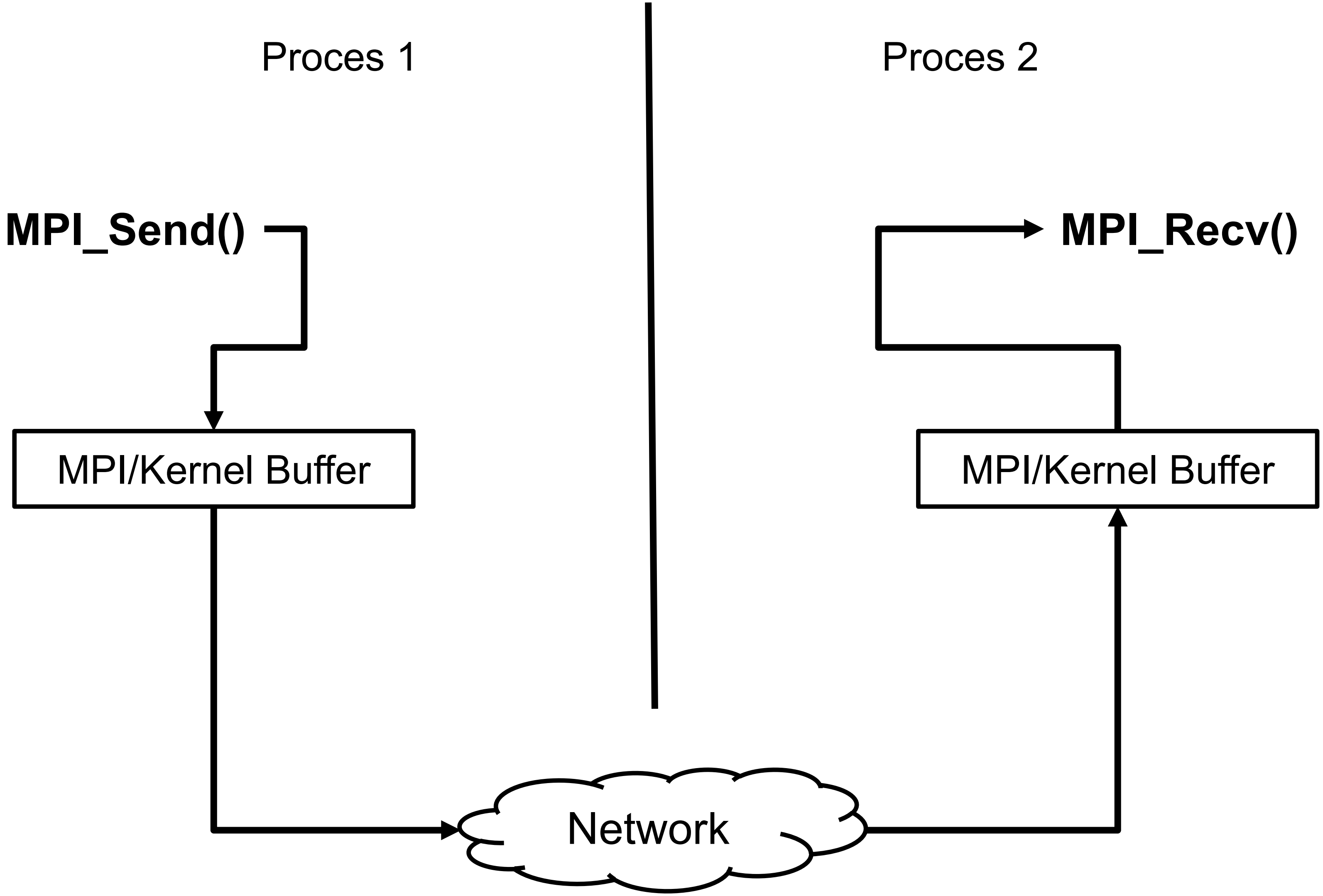
Proces 1

Proces 2

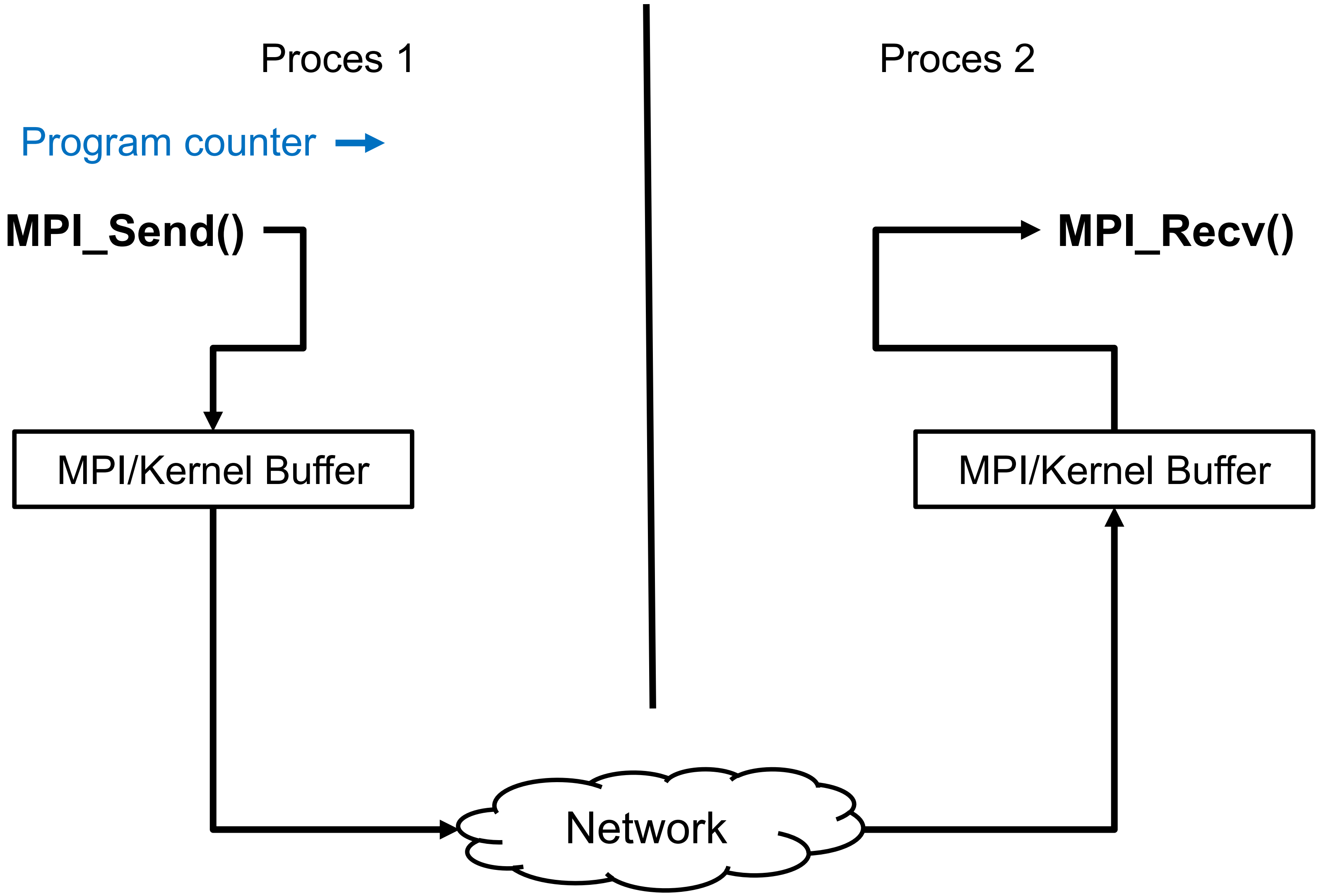
MPI_Send()

MPI_Recv()

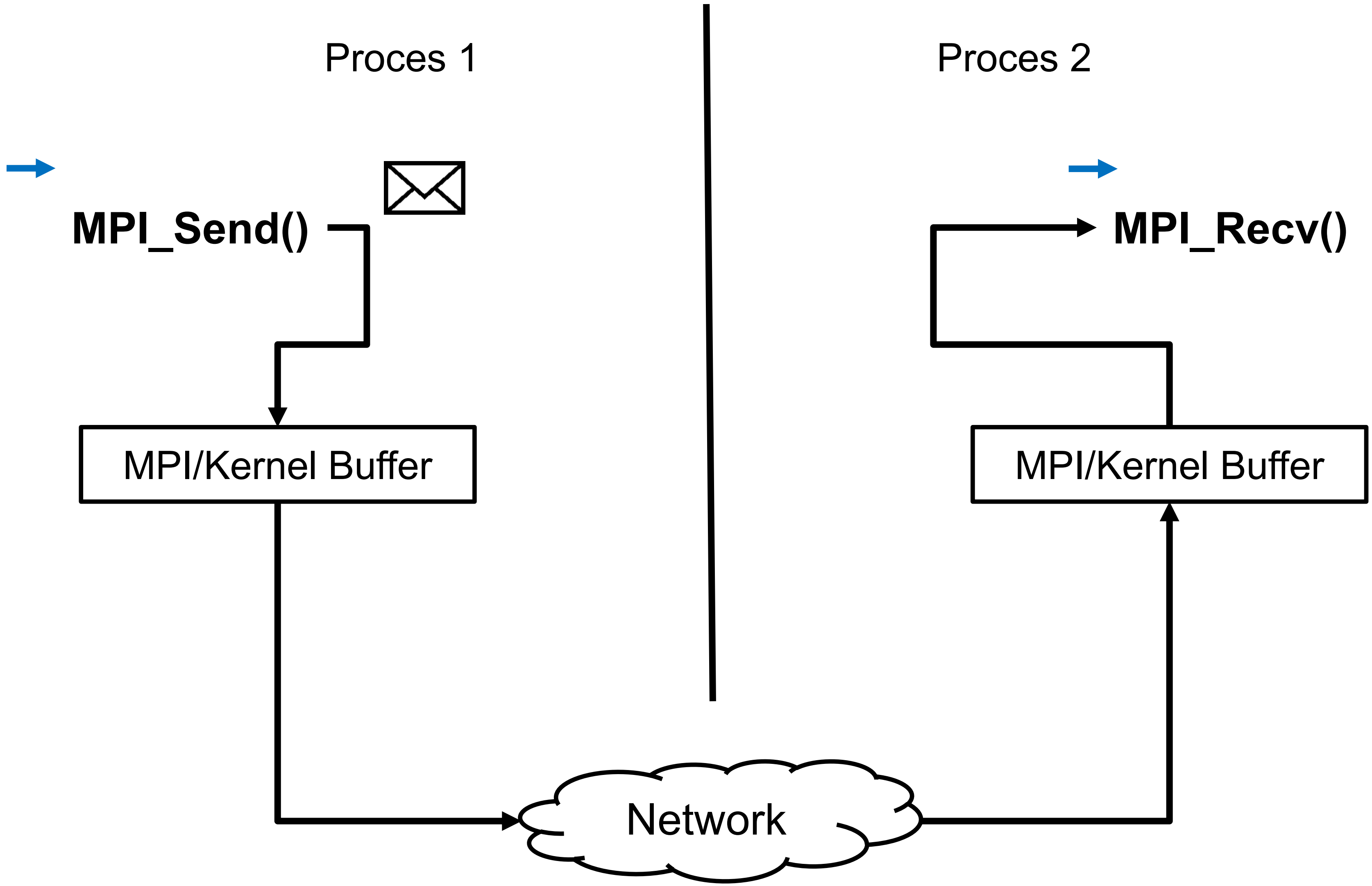
MPI blocking/non-blocking send/recv



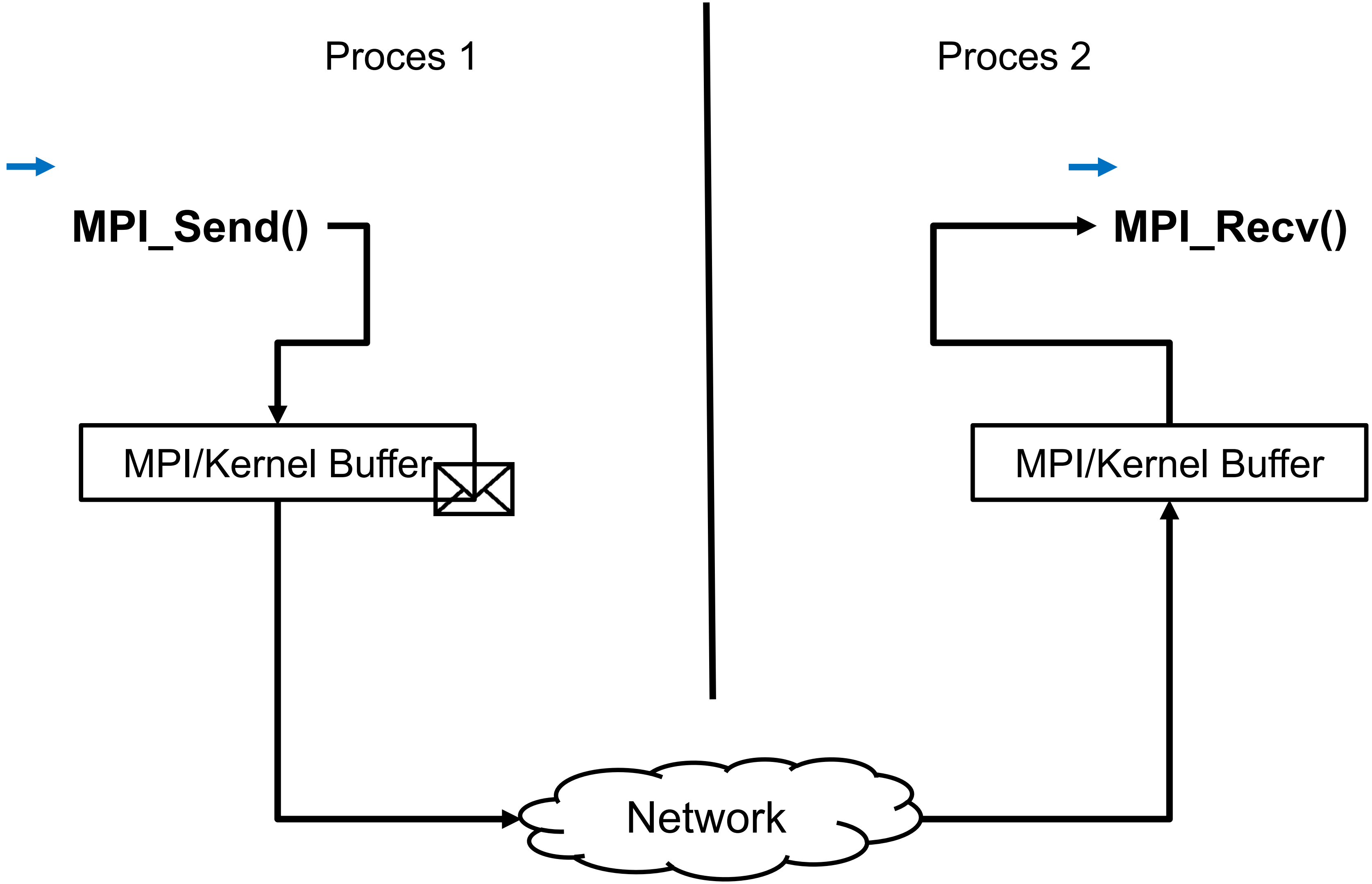
MPI blocking recv/non-blocking send



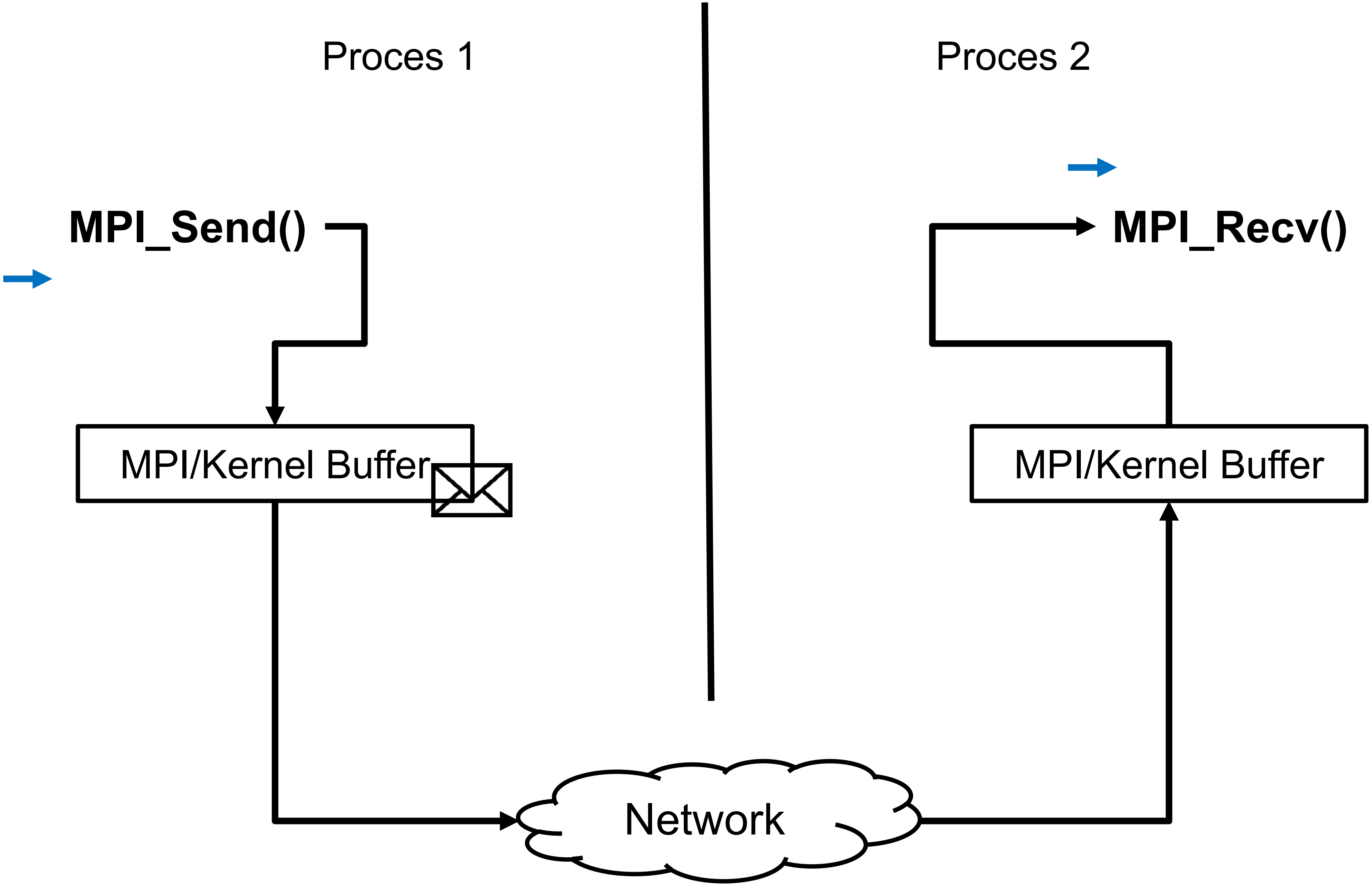
MPI blocking recv/non-blocking send



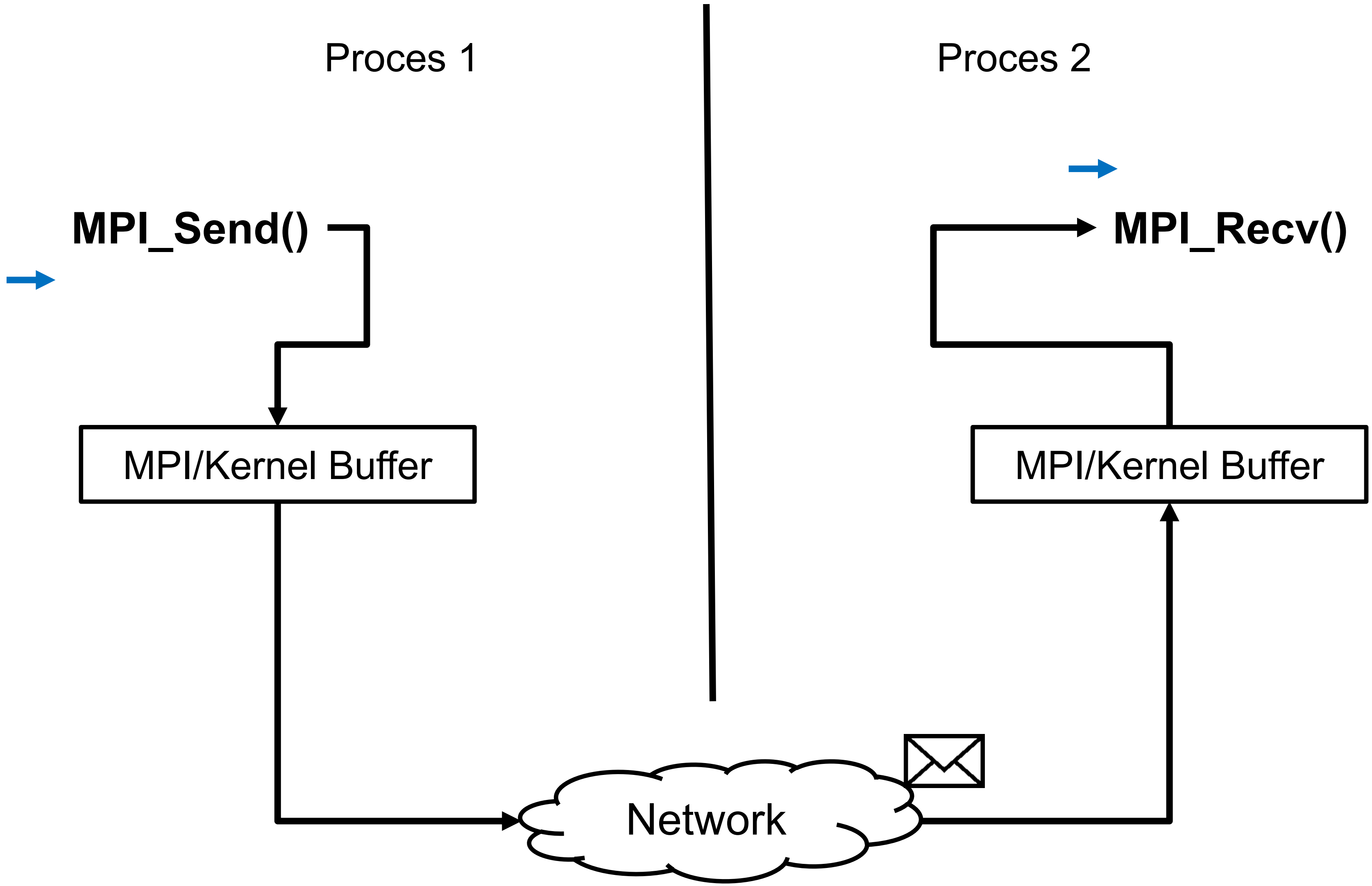
MPI blocking recv/non-blocking send



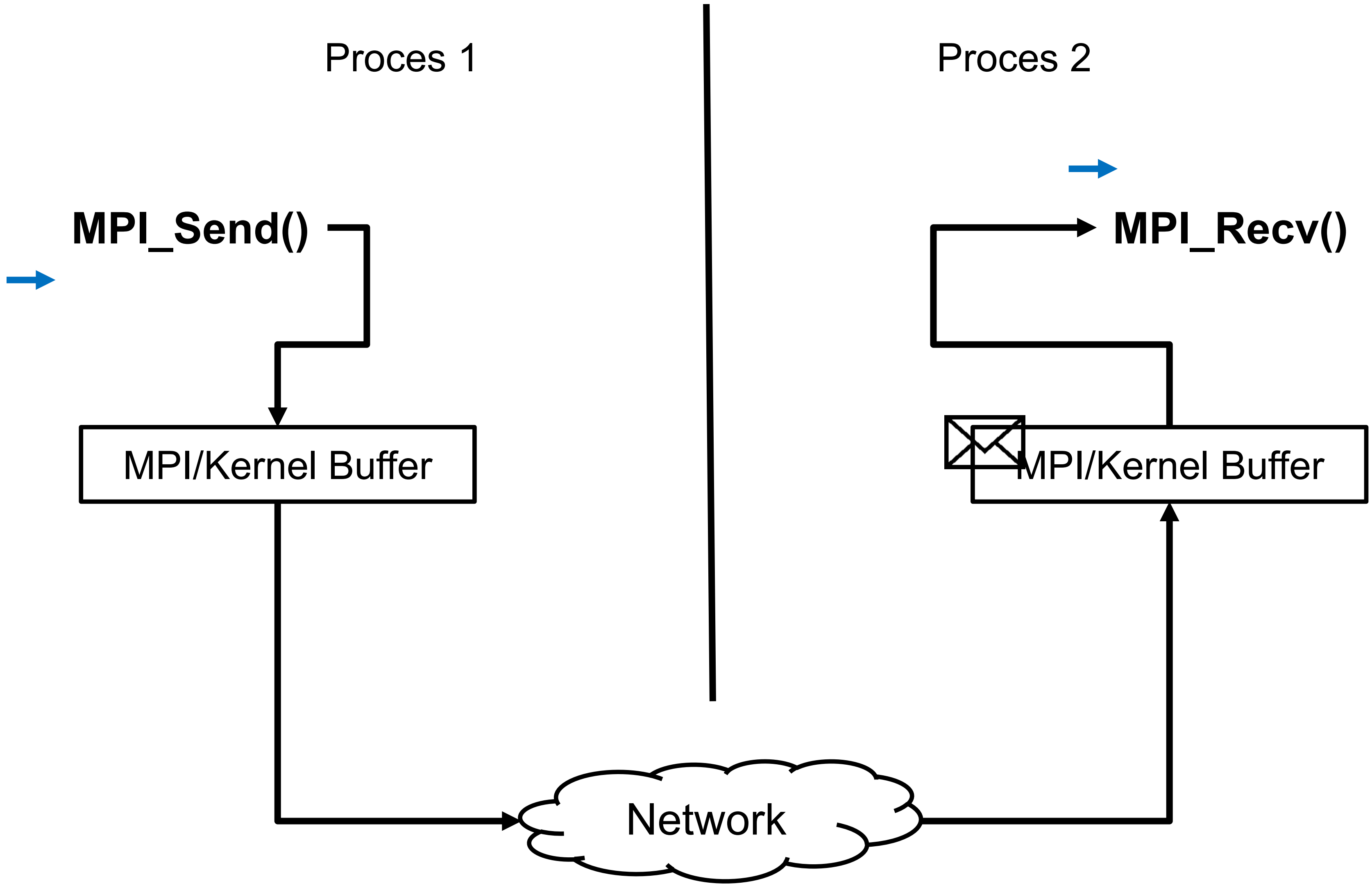
MPI blocking recv/non-blocking send



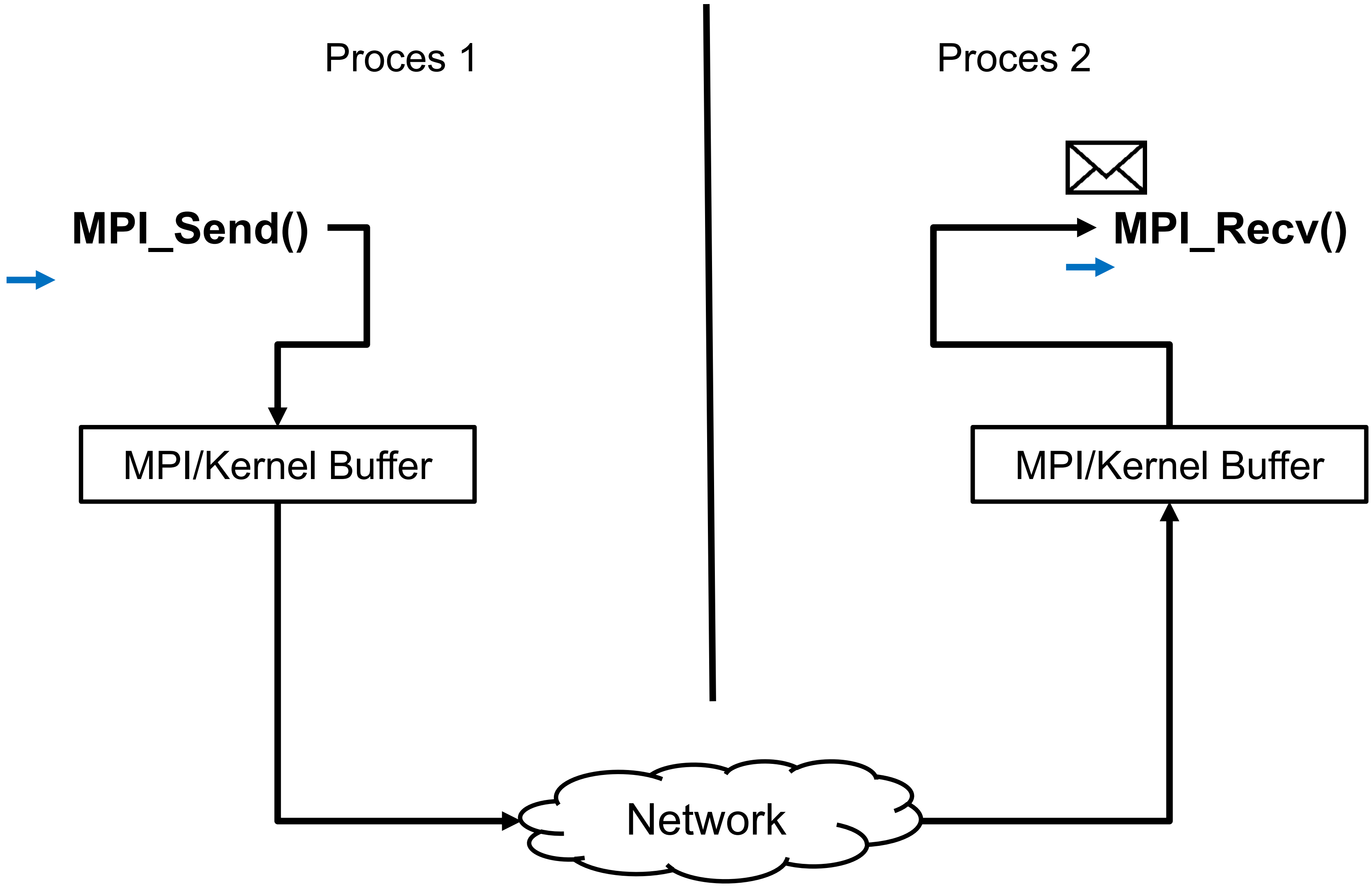
MPI blocking recv/non-blocking send



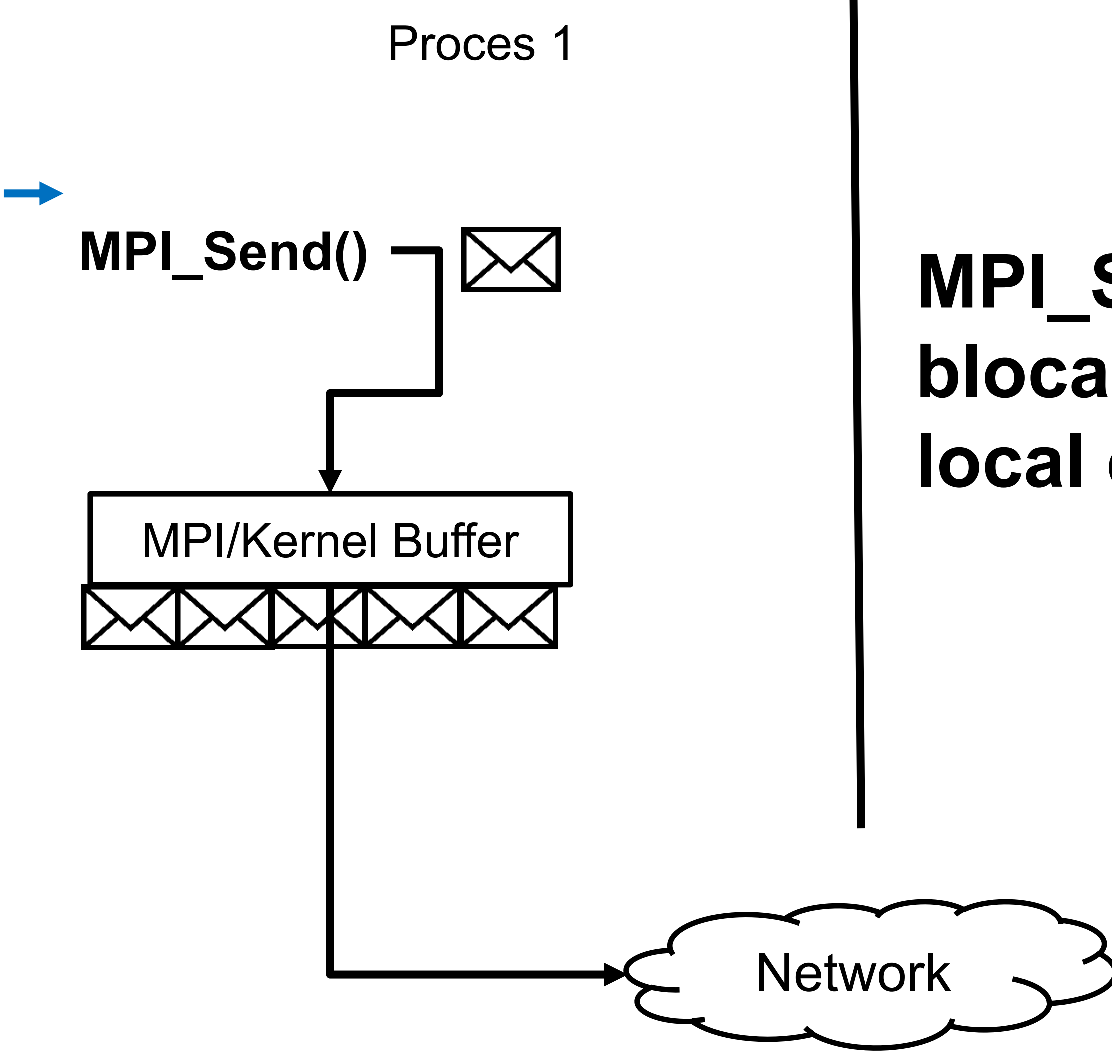
MPI blocking recv/non-blocking send



MPI blocking recv/non-blocking send

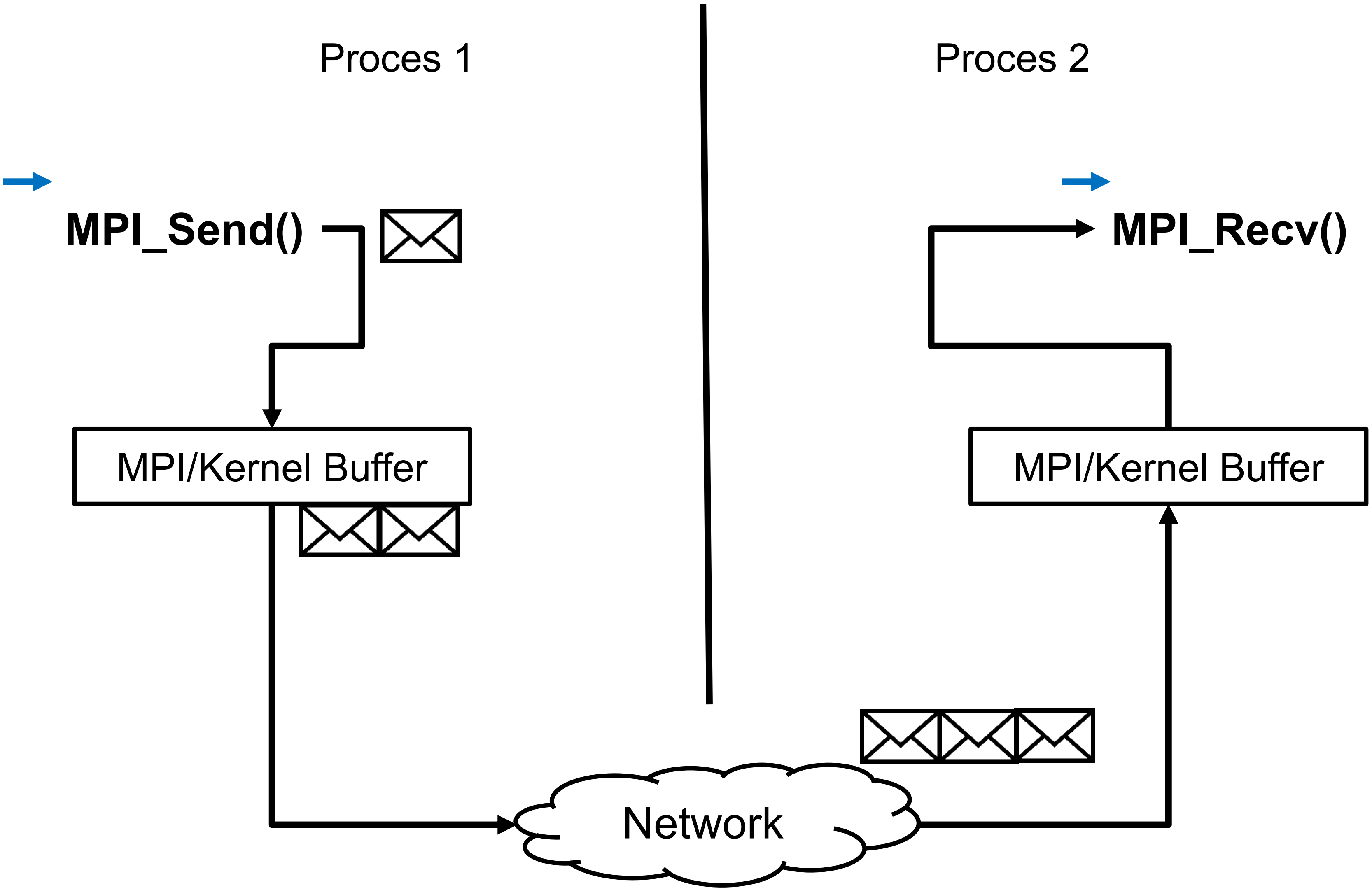


MPI blocking recv/blocking send

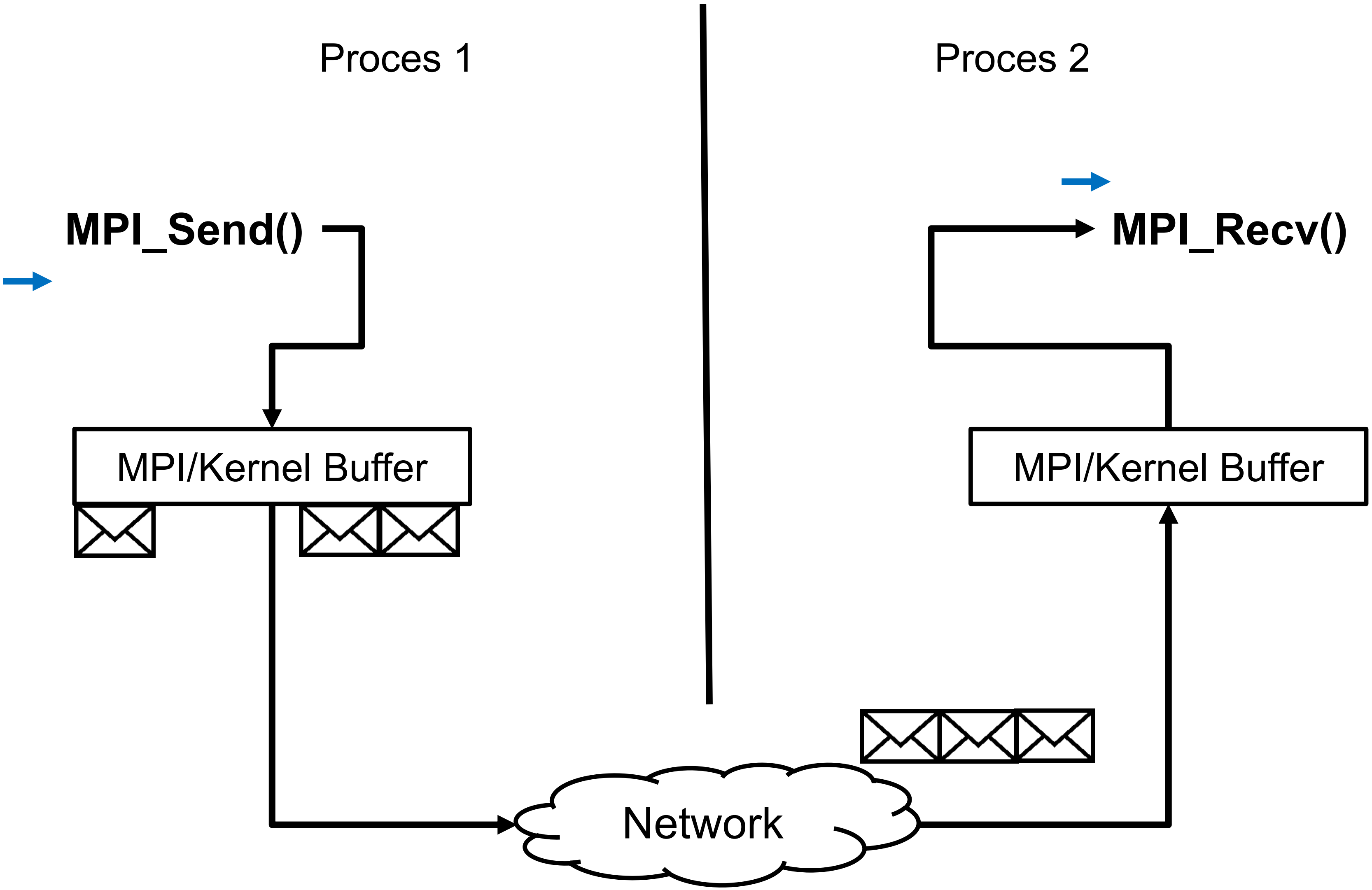


MPI_Send() poate bloca dacă buffer-ul local este plin.

MPI blocking recv/blocking send

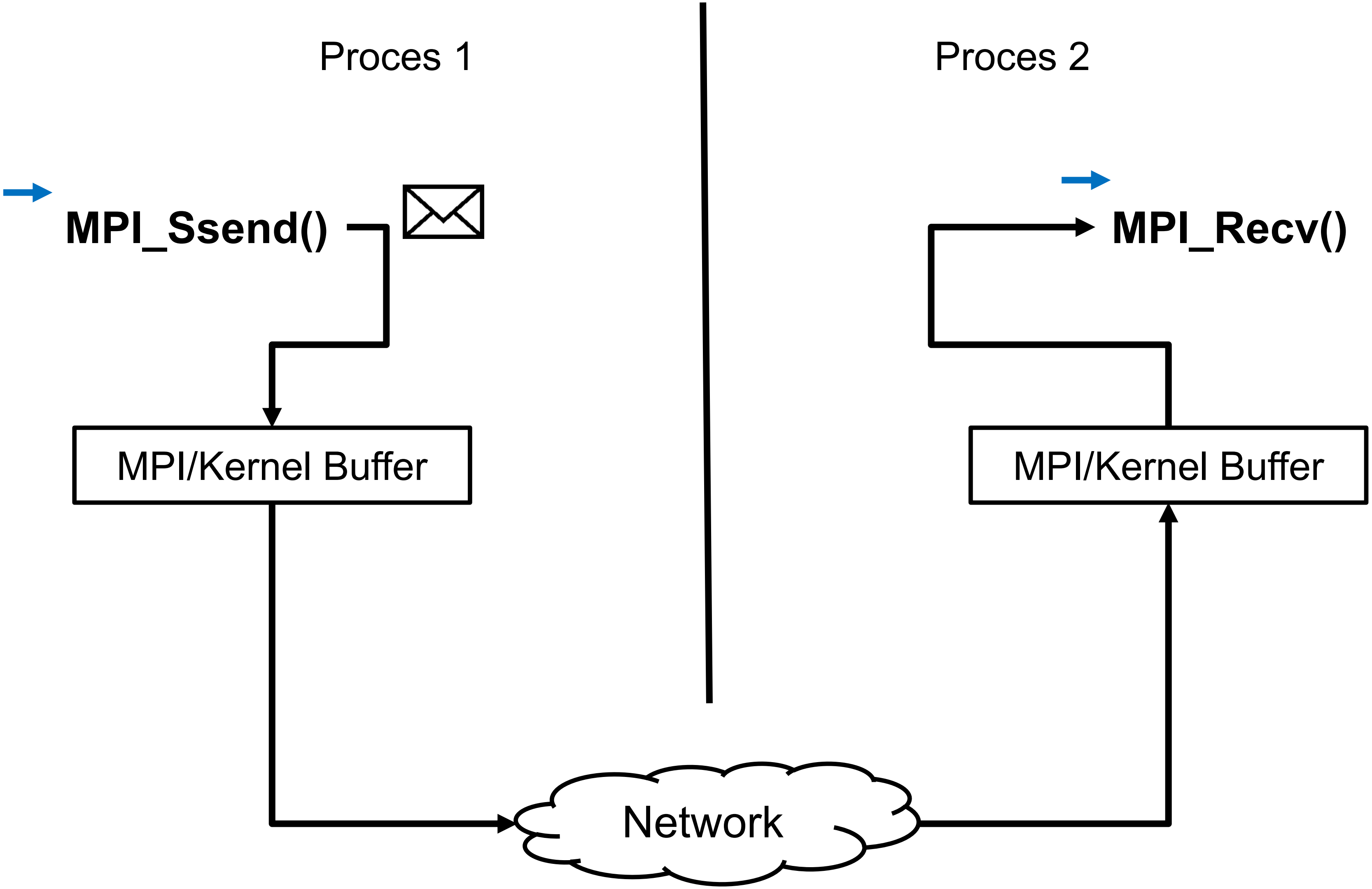


MPI blocking recv/blocking send

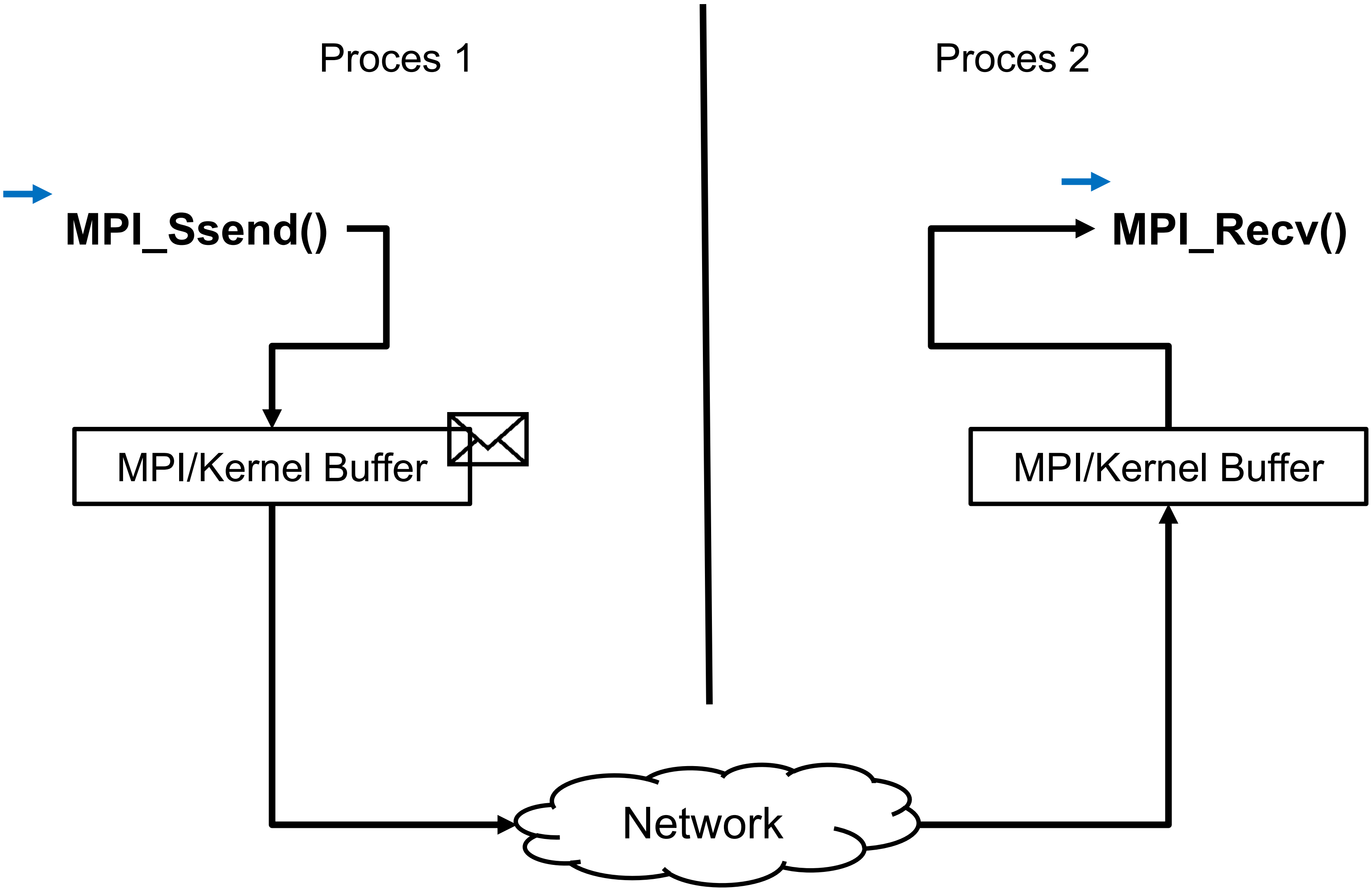


MPI blocking recv/synchronized send

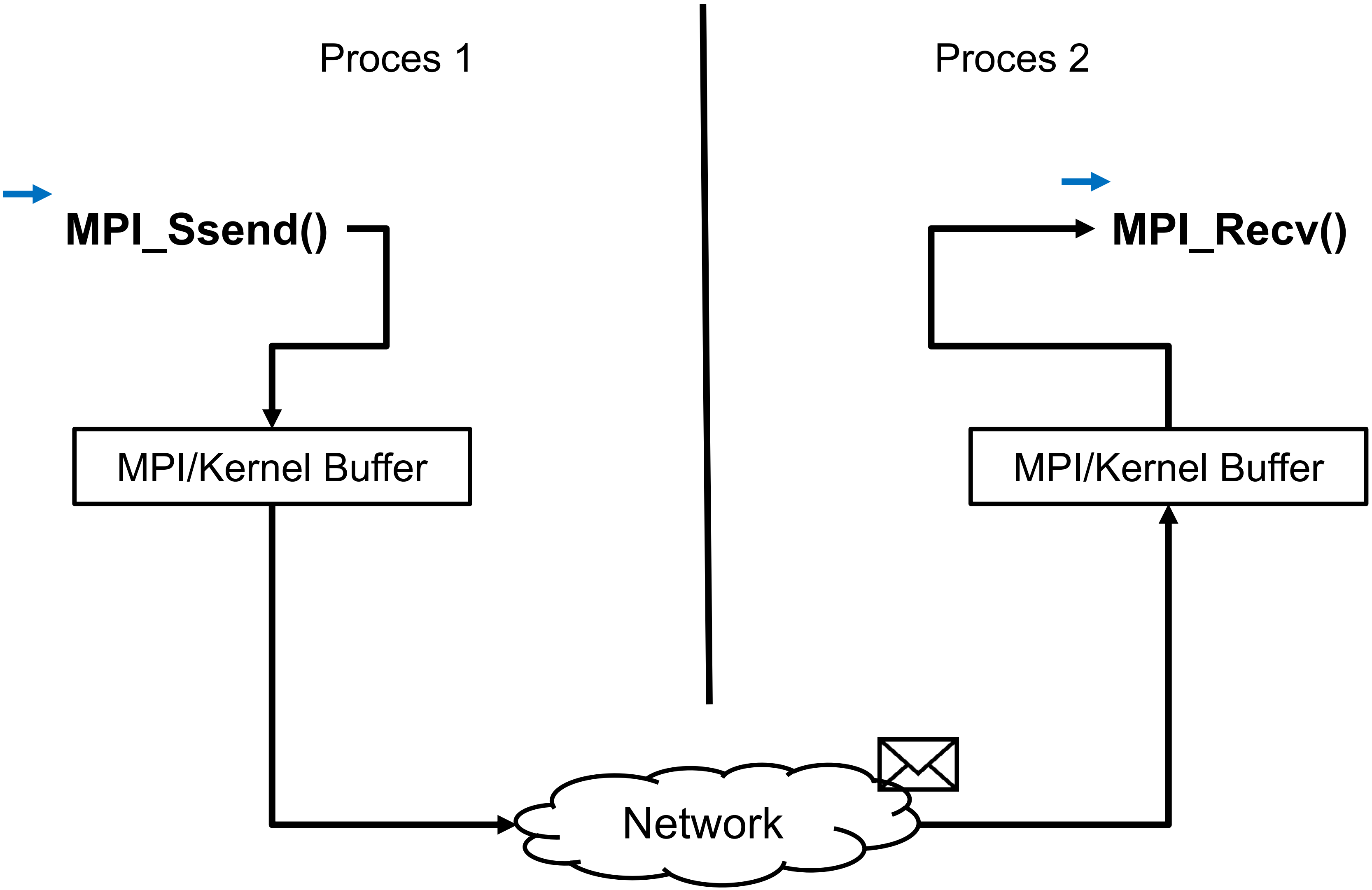
MPI blocking recv/synchronized send



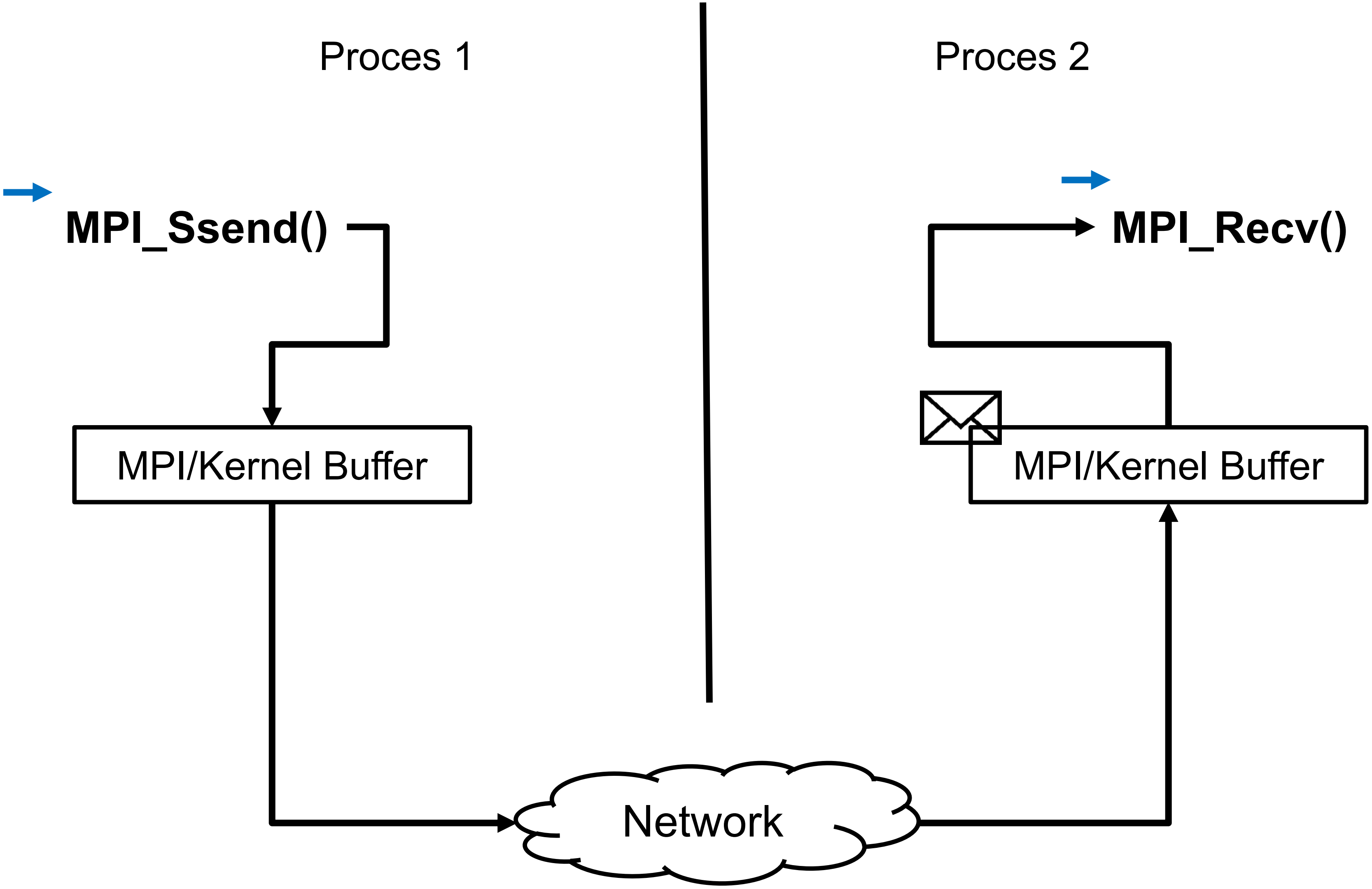
MPI blocking recv/synchronized send



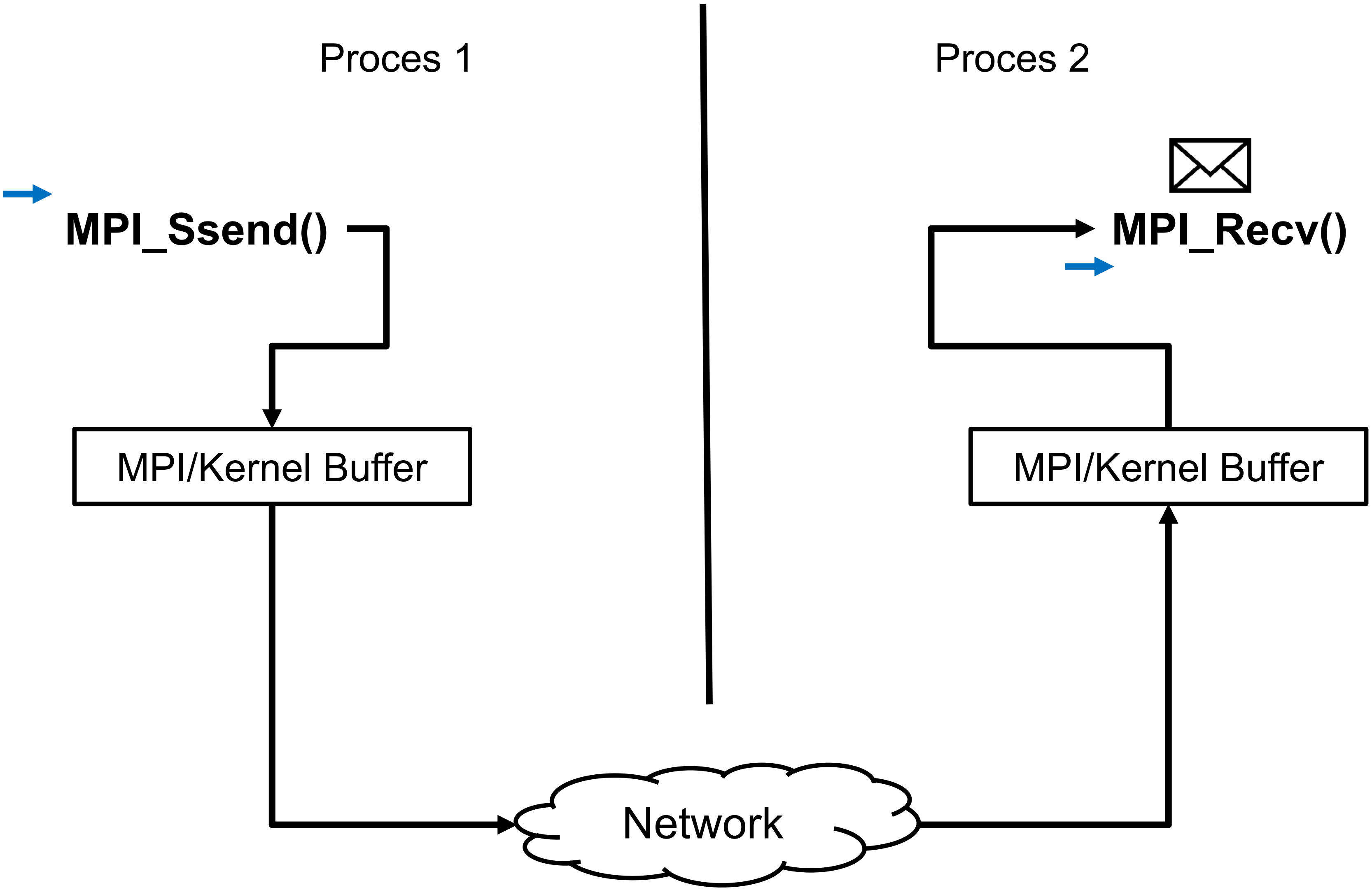
MPI blocking recv/synchronized send



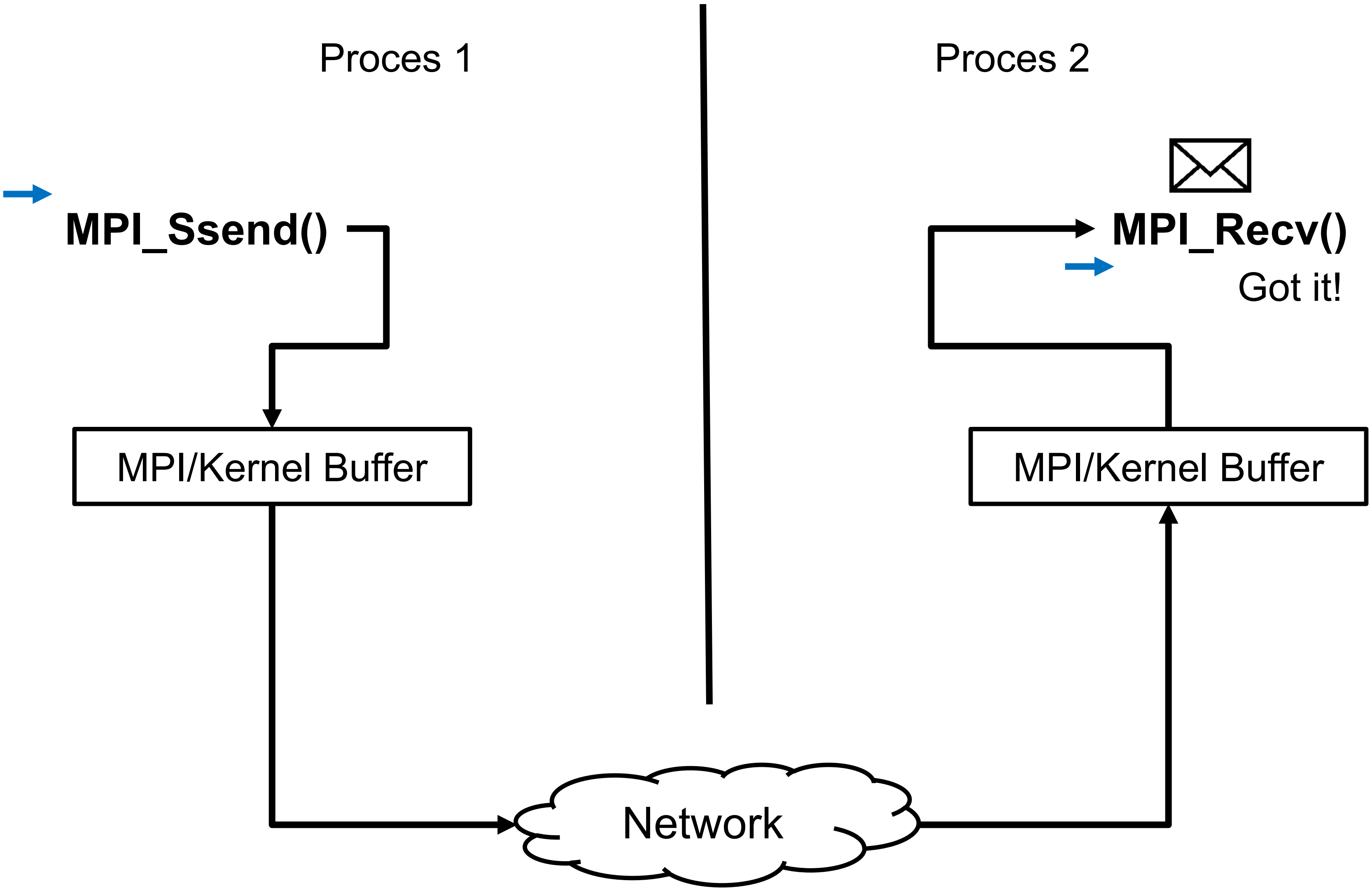
MPI blocking recv/synchronized send



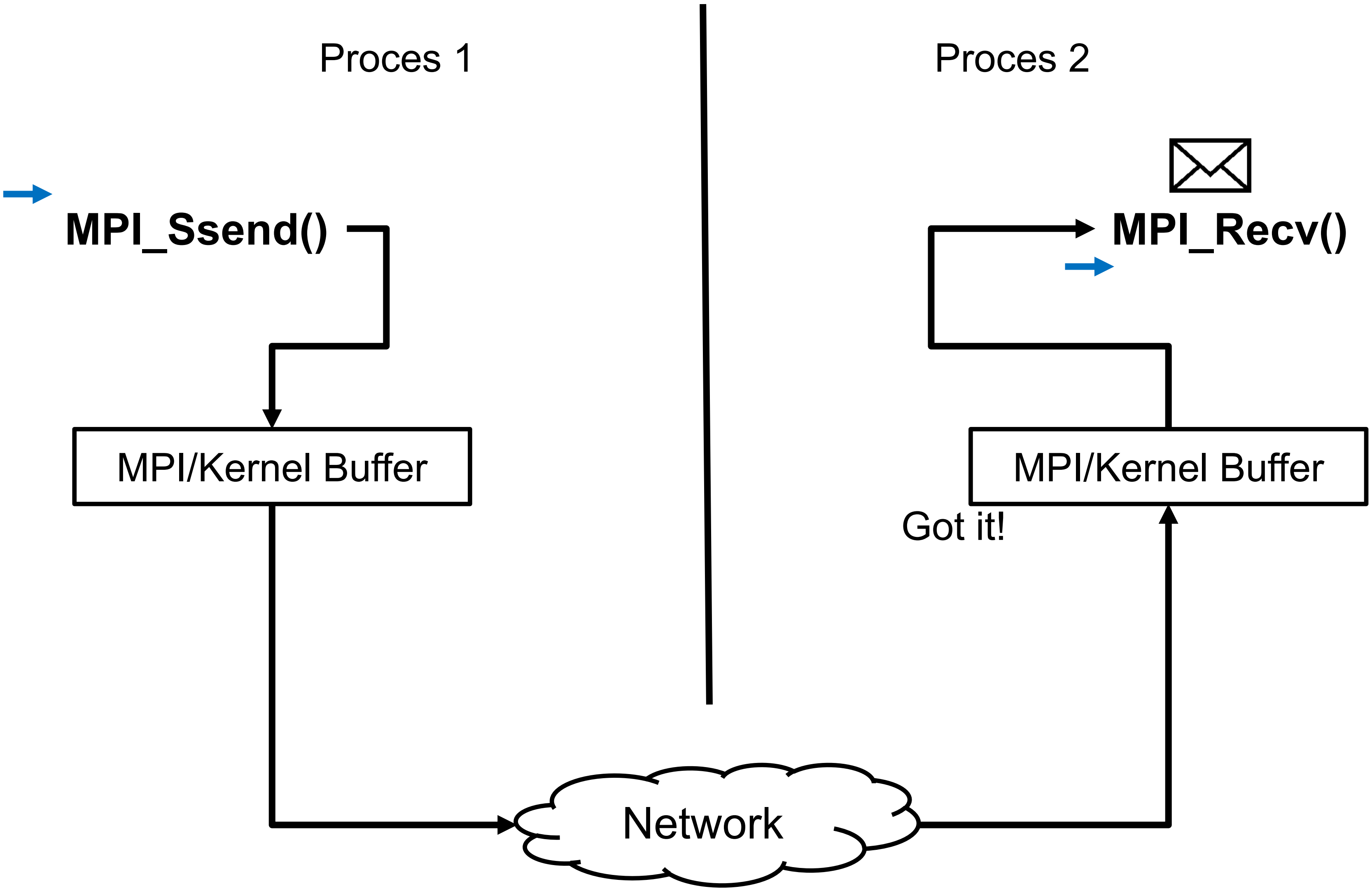
MPI blocking recv/synchronized send



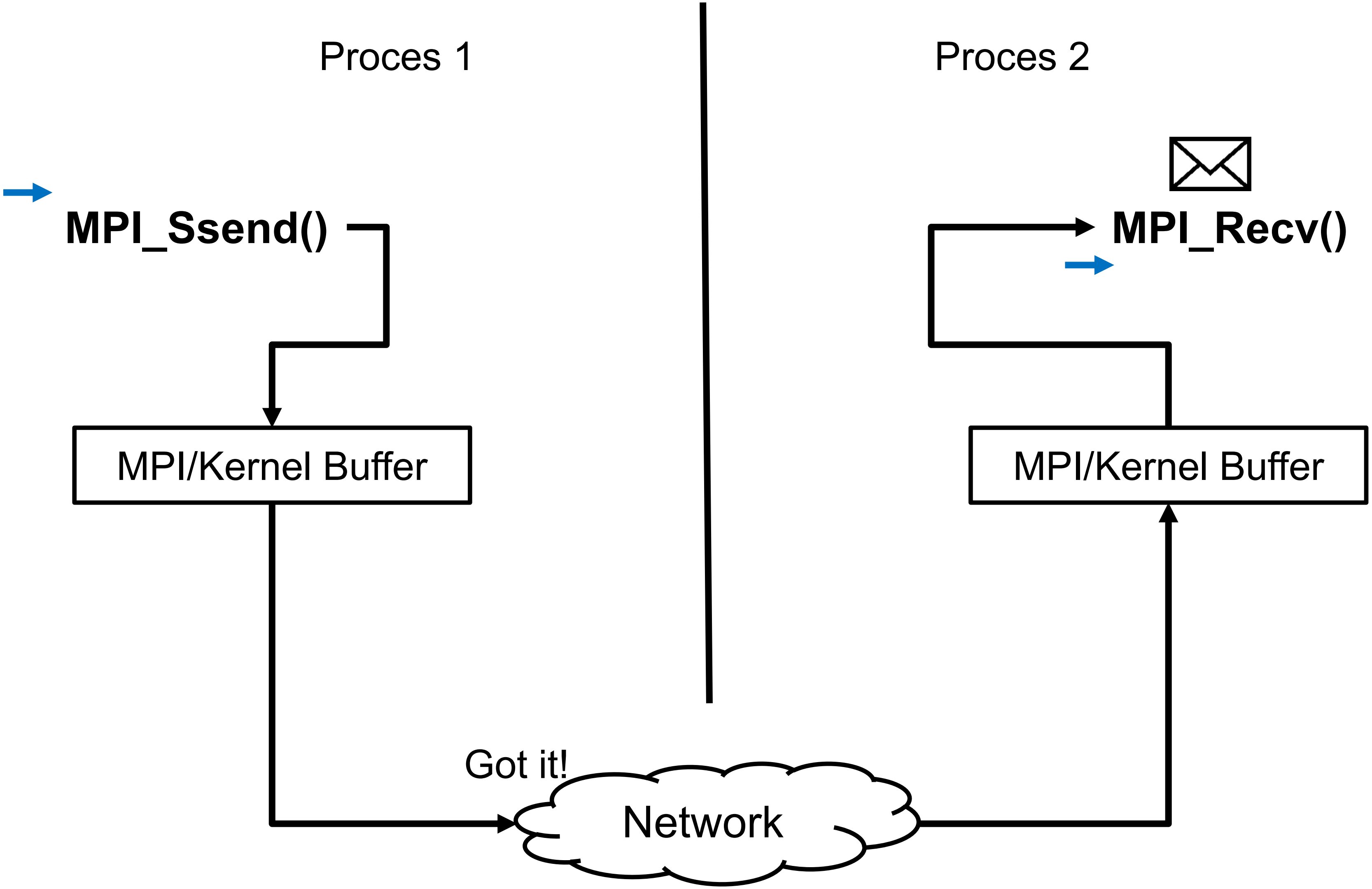
MPI blocking recv/synchronized send



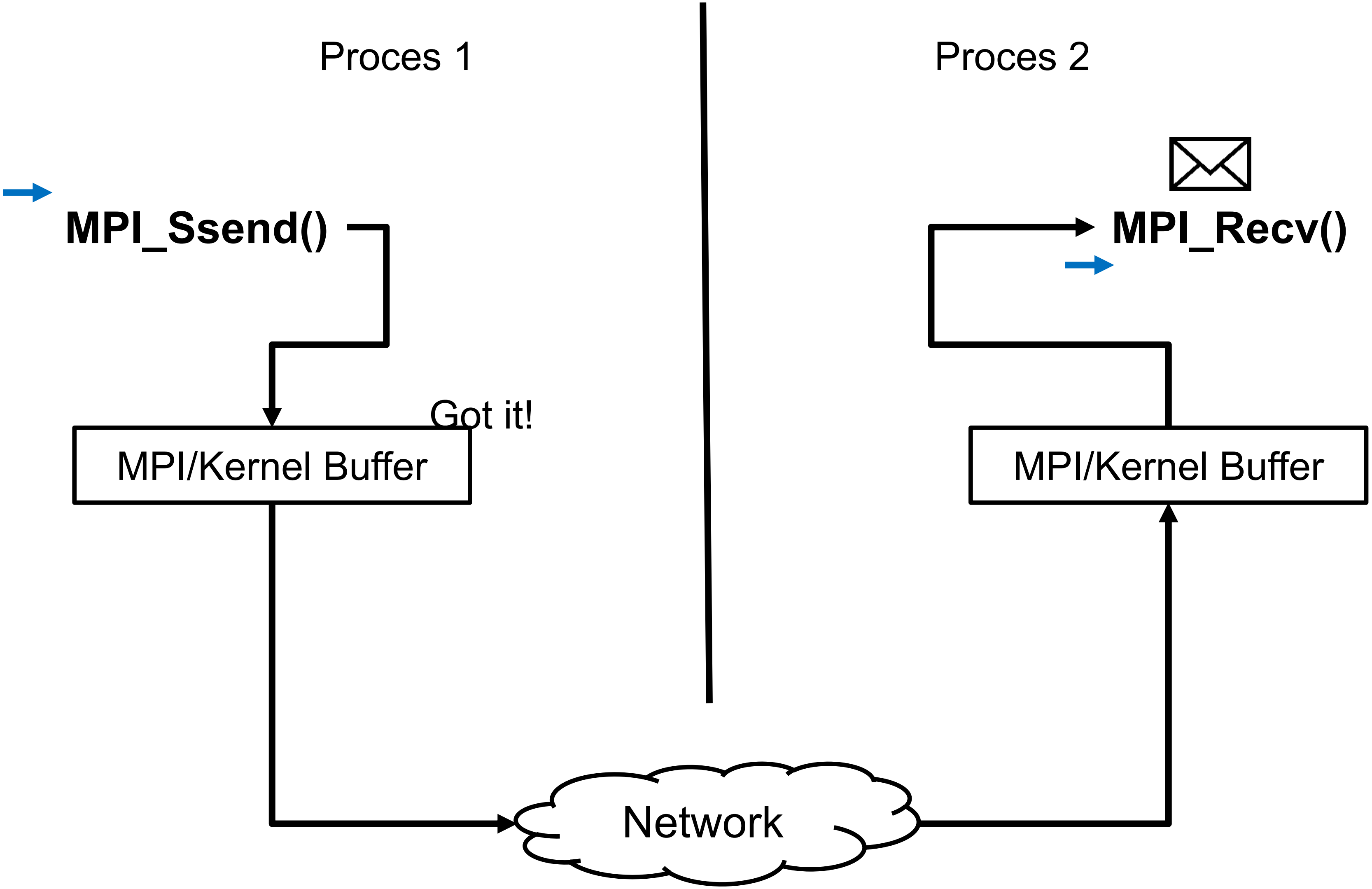
MPI blocking recv/synchronized send



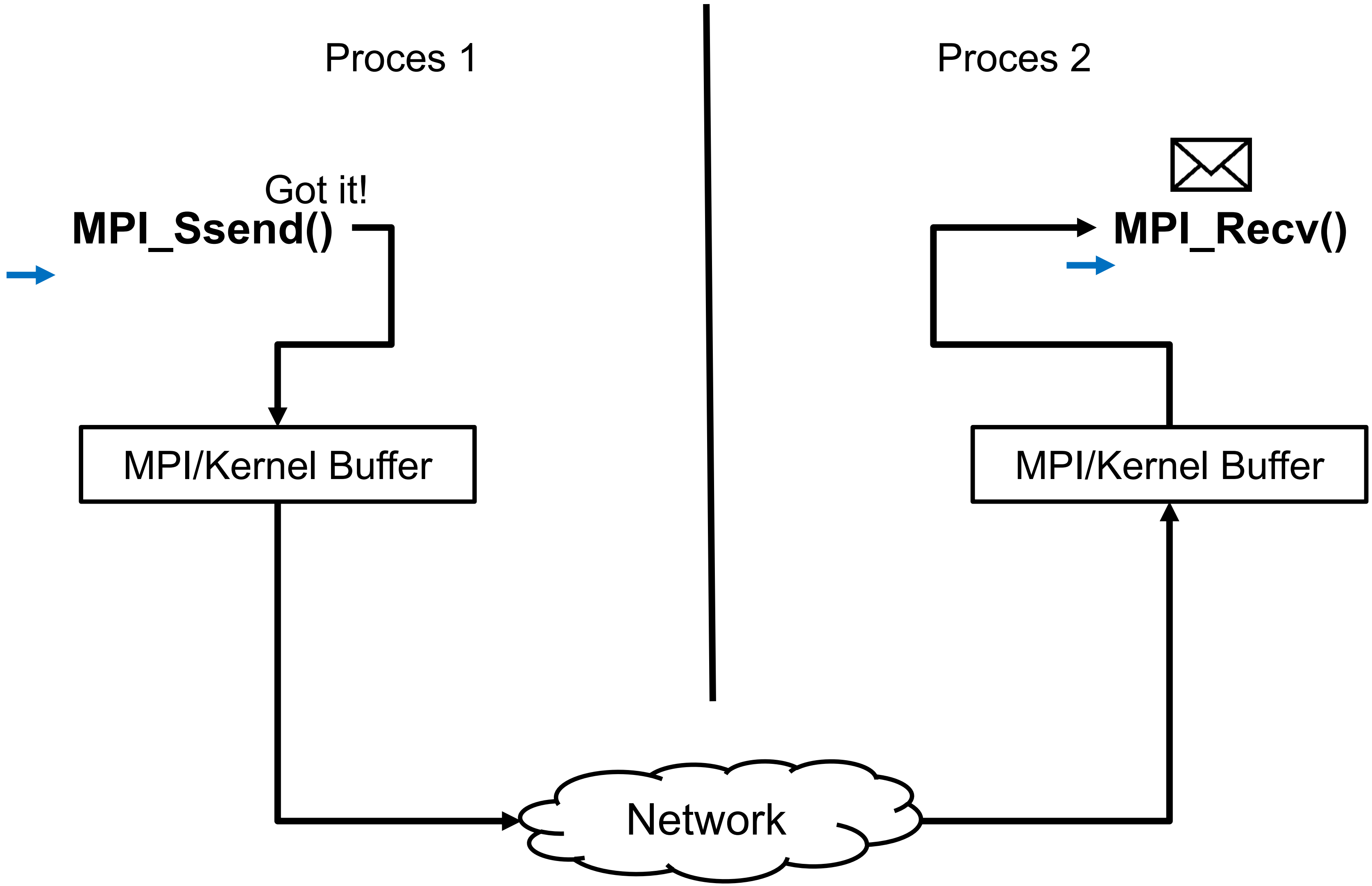
MPI blocking recv/synchronized send



MPI blocking recv/synchronized send

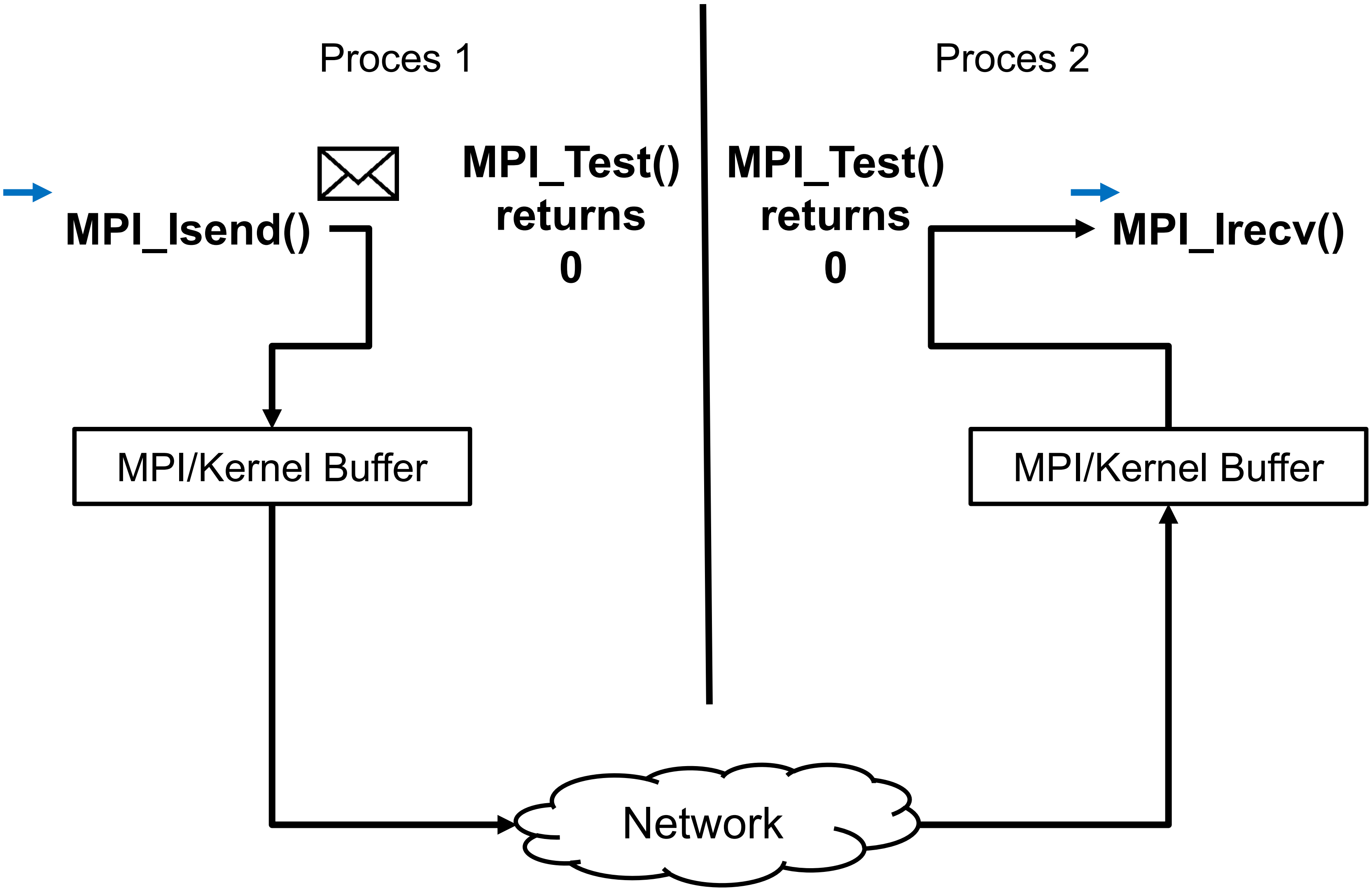


MPI blocking recv/synchronized send

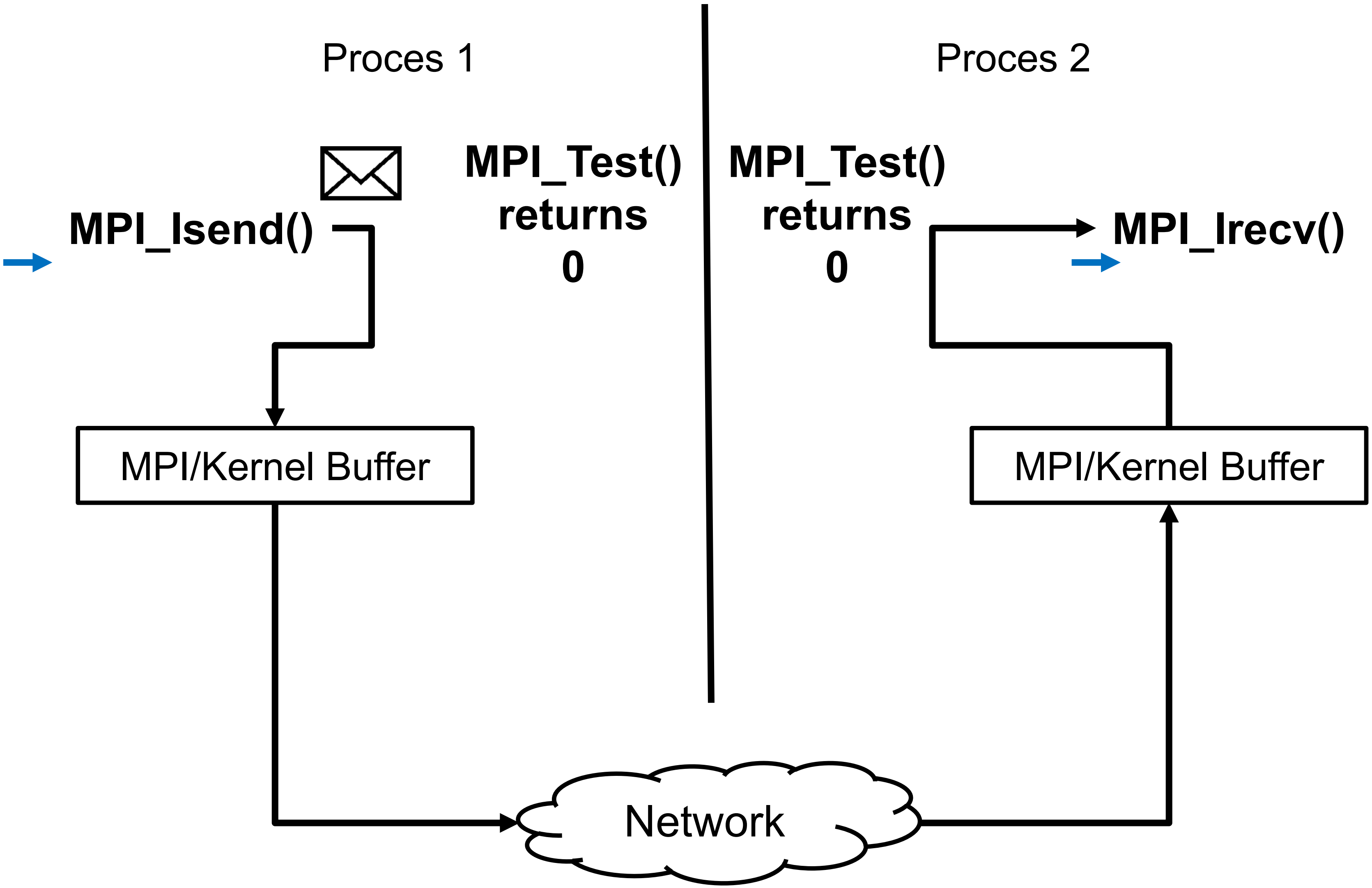


MPI non-blocking recv/non-blocking send

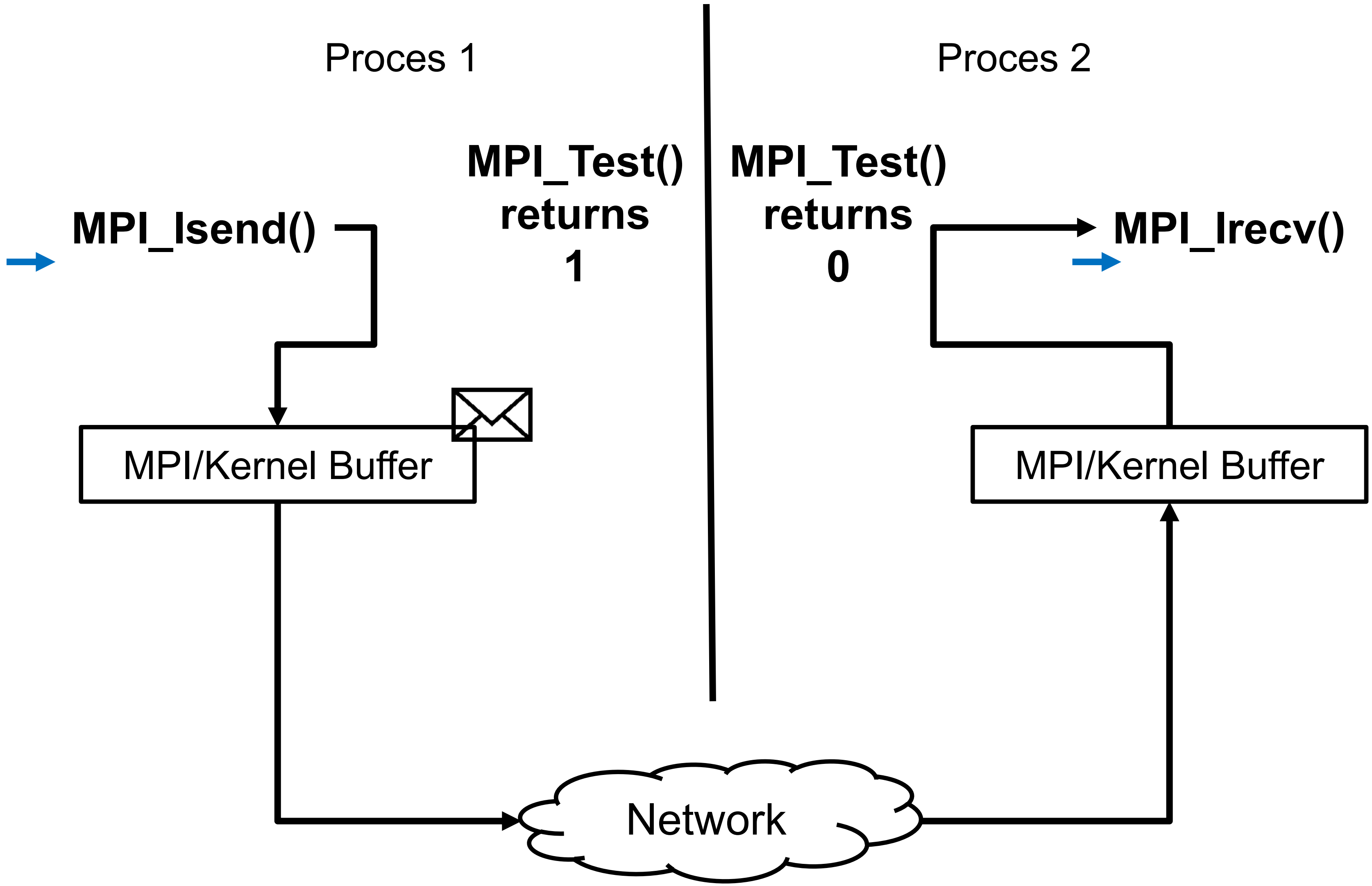
MPI non-blocking recv/non-blocking send



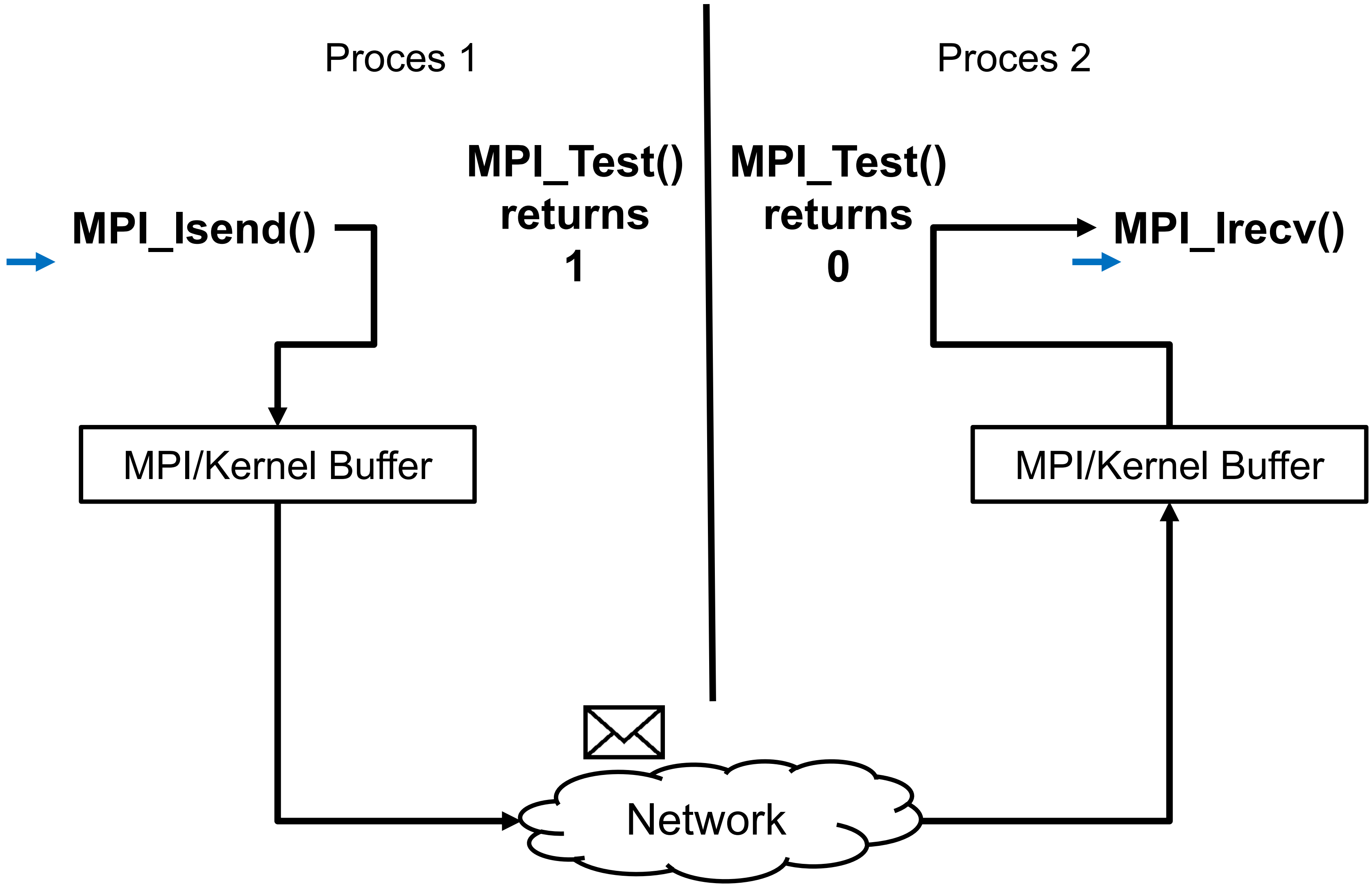
MPI non-blocking recv/non-blocking send



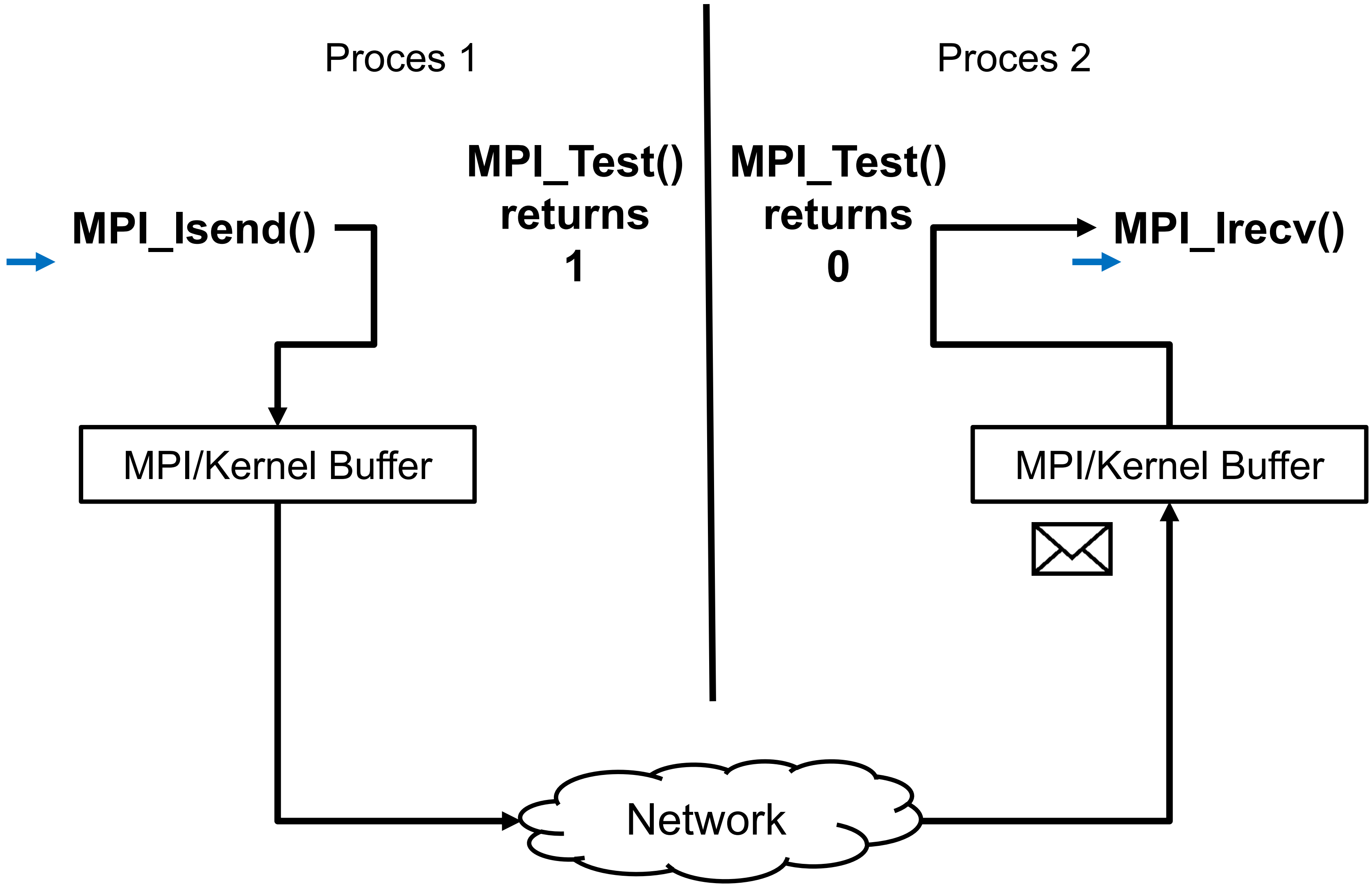
MPI non-blocking recv/non-blocking send



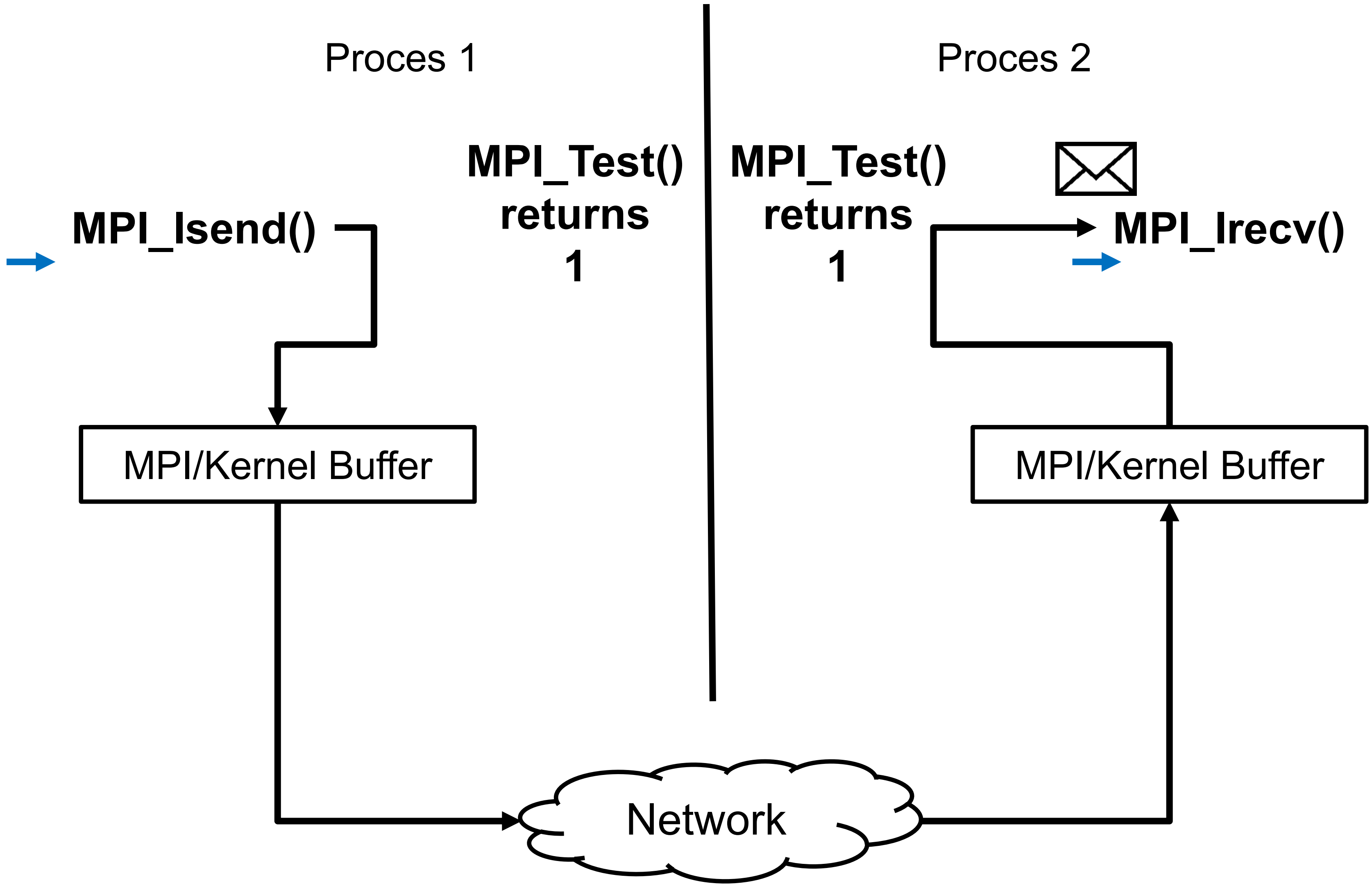
MPI non-blocking recv/non-blocking send



MPI non-blocking recv/non-blocking send



MPI non-blocking recv/non-blocking send



Comunicația non-blocantă

- ```
  int MPI_Isend(void *buf, int count,
            MPI_Datatype datatype, int dest, int tag,
            MPI_Comm comm, MPI_Request *request)
  ```
- ```
  int MPI_Irecv(void *buf, int count,
            MPI_Datatype datatype, int source, int tag,
            MPI_Comm comm, MPI_Request *request)
  ```

- MPI_Test
- MPI_Testall
- MPI_Testany
- MPI_Testsome

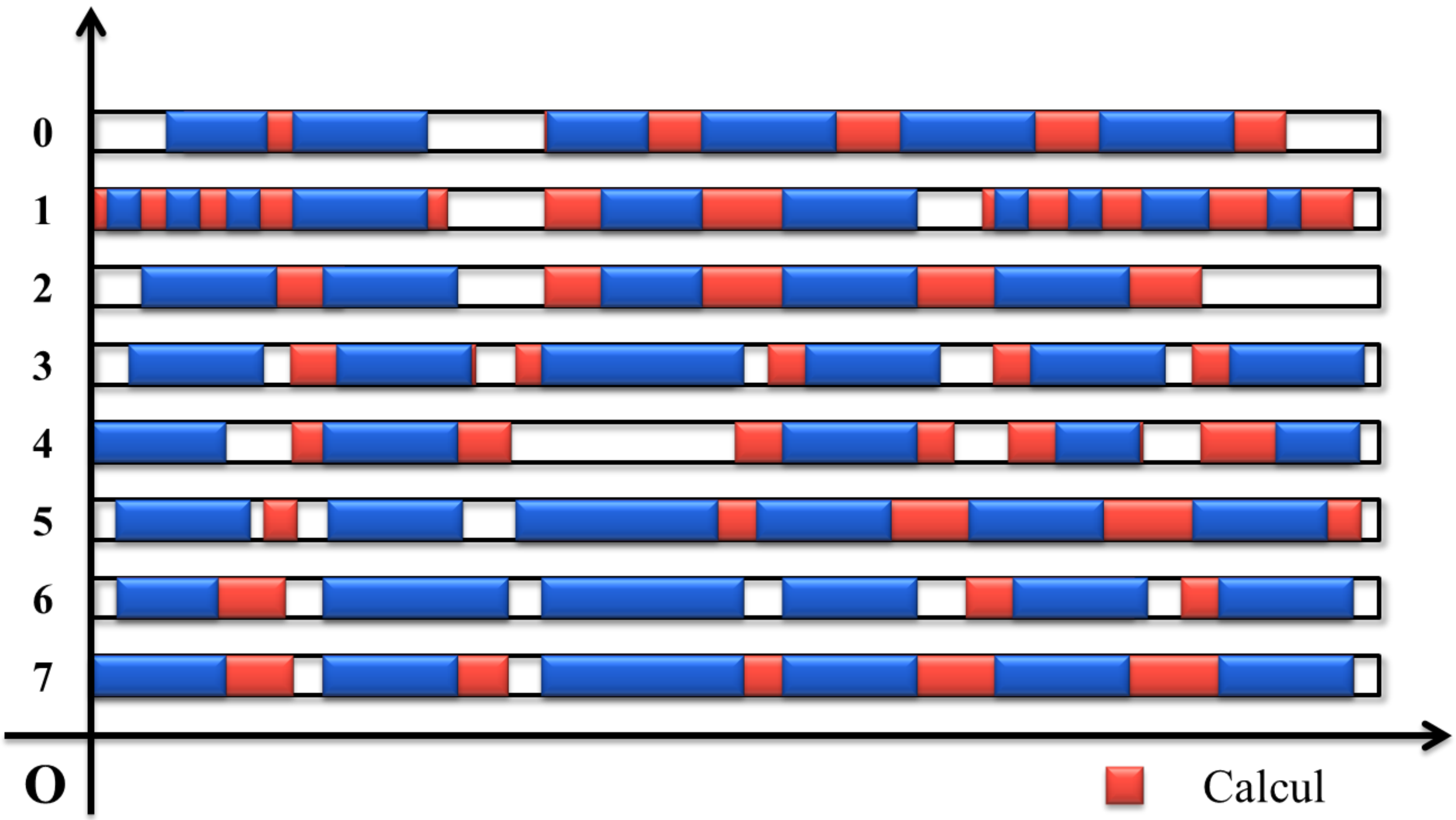
- MPI_Wait
- MPI_Waitall
- MPI_Waitany
- MPI_Waitsome

Modelul Foster

Designing and Building Parallel Programs

Ian Foster

Modelul Foster



(Foster 1995)

Modelul Foster

- Definiţie
  - Timpul scurs de la începerea execuţiei primului proces până la terminarea execuţiei ultimului proces.

$$T = f\,(\,N,\; P,\; U, \dots)$$

$$= T^{j}{}_{comp} + T^{j}{}_{commun} + T^{j}{}_{idle}$$

Unde j un proces arbitrar. SAU

$$\mathrm{T} = \left(\frac{1}{P}\right) * \left(\sum_{i=0}^{P-1} T^{i}{}_{comp} + \sum_{i=0}^{P-1} T^{i}{}_{commun} + \sum_{i=0}^{P-1} T^{i}{}_{idle}\right)$$

$$= \left(\frac{1}{P}\right) * \left(T_{comp} + T_{commun} + T_{commun}\right)$$

LogP model

LogP: Towards a Realistic Model of Parallel Computation *

David Culler, Richard Karp,† David Patterson,
Abhijit Sahay, Klaus Erik Schauser, Eunice Santos,
Ramesh Subramonian, and Thorsten von Eicken

Computer Science Division,
University of California, Berkeley

Abstract

A vast body of theoretical research has focused either on overly simplistic models of parallel computation, notably the PRAM, or overly specific models that have few representatives in the real world. Both kinds of models encourage exploitation of formal loopholes, rather than rewarding development of techniques that yield performance across a range of current and future parallel machines. This paper offers a new parallel machine model, called LogP, that reflects the critical technology trends underlying parallel computers. It is intended to serve as a basis for developing fast, portable parallel algorithms and to offer guidelines to machine designers. Such a model must strike a balance between detail and simplicity in order to reveal important bottlenecks without making analysis of interesting problems intractable. The model is based on four parameters that specify abstractly the computing bandwidth, the communication bandwidth, the communication delay, and the efficiency of coupling communication and computation. Portable parallel algorithms typically adapt to the machine configuration, in terms of these parameters. The utility of the model is demonstrated through examples that are implemented on the CM-5.

David Culler

David Patterson

LogP model

L – Limita superioară a **latenței (latency)** sau întârzierea de transmitere a unui mesaj de la sursă la destinație

o - **overhead**, durata de timp în care procesorul execută transmiterea sau recepţia fiecărui mesaj; În acest timp procesorul nu poate efectua alte operații

g - **gap**, intervalul minim de timp între două transmiteri succesive sau două recepţii succesive la acelaşi procesor. Reciproca lui g este echibalentă cu **lungimea de bandă (bandwidth)**

P - numărul de module **procesor / memorie**. Presupunem că funcționează la aceeași unitate de timp, numită ciclu.

Parțial – la curs 23 Nov

- **Oficiu**
  - 0.1 puncte
- **Grilă 35 întrebări în 20 de minute – pe moodle**
  - 1.4 puncte
  - Un singur răspuns corect din 4.
- **O problemă submisă pe checker**
  - 0.5 puncte
  - 45 de minute (se pot primi punctaje parțiale)
  - După 45 de minute se notează primii 10 care au submis implementare cu punctaj maxim (limitat la o oră jumate)

Parțial

- **Necesar 1 punct din 2 pentru promovare.**
- **Se poate reda o dată cu examenul final.**
- **Dacă este promovat degrevează materia de programare paralelă din examenul final.**
- **Se poate reda și pentru mărire o dată cu examenul final (cu riscul de a micșora nota).**