# An Introduction to MPI Parallel Programming with Java

Xuguang Chen
Department of Computer Science
Saint Martin's University
Lacey, WA,

# What is MPI?

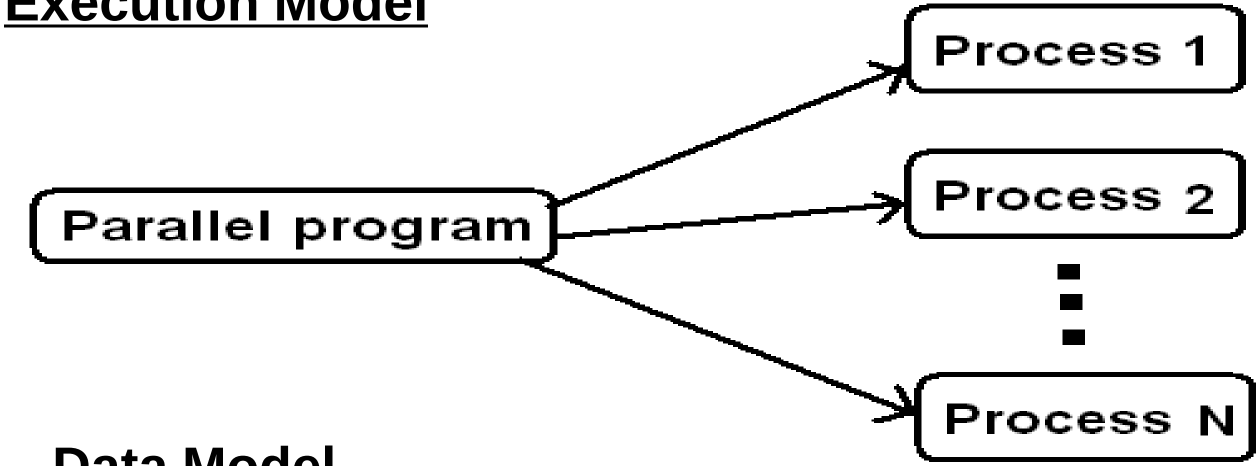
- Message Passing Interface (MPI)
  - MPI is a specification for the developers and users of message passing libraries.
  - By itself, it is NOT a library - but rather the specification of what such a library should be.
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-4.0
  - https://www.mpi-forum.org/
- Actual MPI library implementations differ in which version and features of the MPI standard they support.
- MPI primarily *addresses the message-passing parallel programming model*
  - Data is moved from the address space of one process to that of another process through cooperative operations on each process.

# Background Information

- Execution model in MPI
  - Parallel program is launched as set of independent, identical processes
  - All the processes contain the same program code and instructions
  - Processes can reside in different nodes or even in different computers
  - The way to launch parallel program is implementation dependent
    - e.g., mpirun, mpiexec
  - There are two classes of message passing (transfers)
    - Point-to-Point messages involve only two tasks
    - Collective messages involve a set of tasks
- Data model in MPI
  - All variables and data structures are local to the process
  - Processes can exchange data by sending and receiving messages

# Background Information

## Execution Model



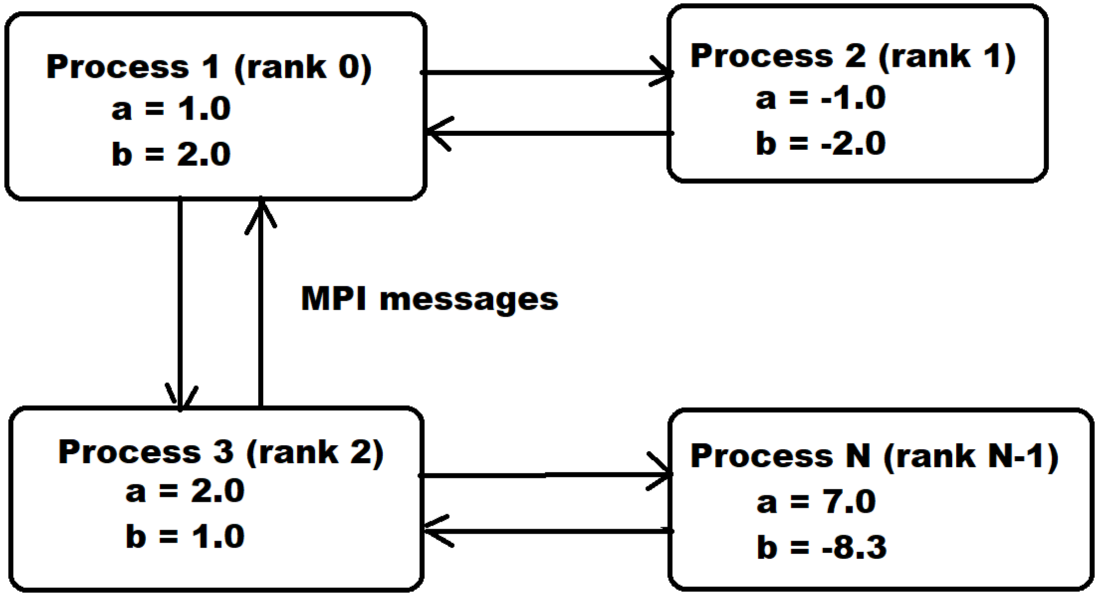
Parallel program

Process 1

Process 2

Process N

## Data Model



**Process 1 (rank 0)**
**a = 1.0**
**b = 2.0**

**Process 2 (rank 1)**
**a = -1.0**
**b = -2.0**

**MPI messages**

**Process 3 (rank 2)**
**a = 2.0**
**b = 1.0**

**Process N (rank N-1)**
**a = 7.0**
**b = -8.3**

# Background Information

- MPJ Express
  - An open source Java message passing library
  - It allows application developers to write and execute parallel applications for multicore processors and compute clusters/clouds.
  - It is distributed under the MIT (a variant of the LGPL) license.
  - Cite (reference) the following paper for acknowledging the use of MPJ Express in research work.
    - Aamir Shafi, Bryan Carpenter, Mark Baker, Nested parallelism for multi-core HPC systems using Java, Journal of Parallel and Distributed Computing, Volume 69, Issue 6, 2009, Pages 532-545, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2009.02.006.
  - Documents
    - http://www.mpjexpress.org/docs/javadocs/index.html
  - URL
    - http://www.mpjexpress.org/

# Installation and Execution

- Download Software
  - Download and install JDK, such as https://www.oracle.com/java/technologies/downloads/
  - Download MPJ and unzip, such as “mpj-v0_44.zip” https://sourceforge.net/projects/mpjexpress/files/releases/
- Install in Linux
  - Download and unzip
    - Assume that the files are saved in a directory named mpj-v0_44
  - Setting the path and variables
    - *export MPJ_HOME=/home/pi/mpj-v0_44*
    - *export PATH=$MPJ_HOME/bin:$PATH*

# Installation and Execution

- Install in Windows
  - Assume that the zipped file is unzipped in a folder named “*mpj_v0_44*” on the “*D*” drive
  - Setting the path and variables in Windows 11
    - Right click to show “settings” menu
    - Choose “System” on the menu
    - Choose “Advanced system settings”
    - Choose “Environment Variables”
    - Choose “New” under the top section
    - Enter “MPJ_HOME” in Variable name
    - Enter “D:\mpj_v0_44” in Variable value
    - Choose “Path” in the bottom section, and then click “Edit”
    - Choose “New” on “Edit environment variable”, enter “%MPJ_HOME%\bin;”, and click “ ”
  - The path and variables in Windows 10 can similarly be set

# Installation and Execution

- Connect Raspberry PI to a Laptop
  - The raspberry pi needs to be on the same network as the laptop, i.e., both are connected to the same router.
  - Connect the raspberry Pi to the monitor, mouse and keyboard.
  - Using the pi’s terminal:
    - Type: ifconfig
      - If you’re using ethernet, look for the address in the eth0 section
      - If you’re using wifi, look for the address in the wlan0 section.
  - Type sudo halt to shut down the rasperry pi.
  - Open a PUTTY, enter the IP, and select “Enable X11 forwarding”
    - Username is *pi*, and password is *raspberry*.

# Installation and Execution

- Compilation and Execution in Windows
  - Edit source code
    - Many text editors can be used, such as Notepad++, Textpad, Windows Notepad, or NetBeans
  - Compile
    - *javac -cp .;%MPJ_HOME%/lib/mpj.jar file_name.java*
  - Execution
    - *%MPJ_HOME%/bin/mpjrun.bat -np 2 file_name*
- Compilation and Execution in Linux
  - Edit source code
    - Many text editors can be used, such as pico
  - Compile
    - *javac -cp .:$MPJ_HOME/lib/mpj.jar file_name.java*
  - Execution
    - *mpjrun.sh -np 2 file_name*

# Typical MPI Code Structure

MPI Include File

Variable declarations, etc

Begin Program
…
Serial code
….

MPI Initialization

Parallel Code begins

MPI Rank (process identification)

…
Parallel code based on rank
…

MPI Communications between processes

…
Parallel code based on rank
…

MPI Finalize (terminate)

Parallel Code ends

Serial Code

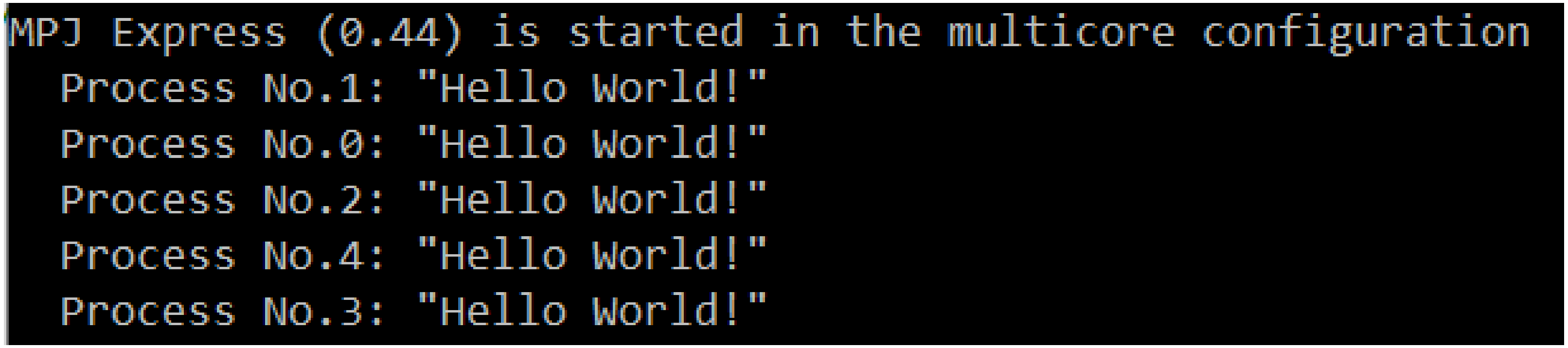
# First Program

- A variation on the standard hello world program - Hello World program for multiple processes.

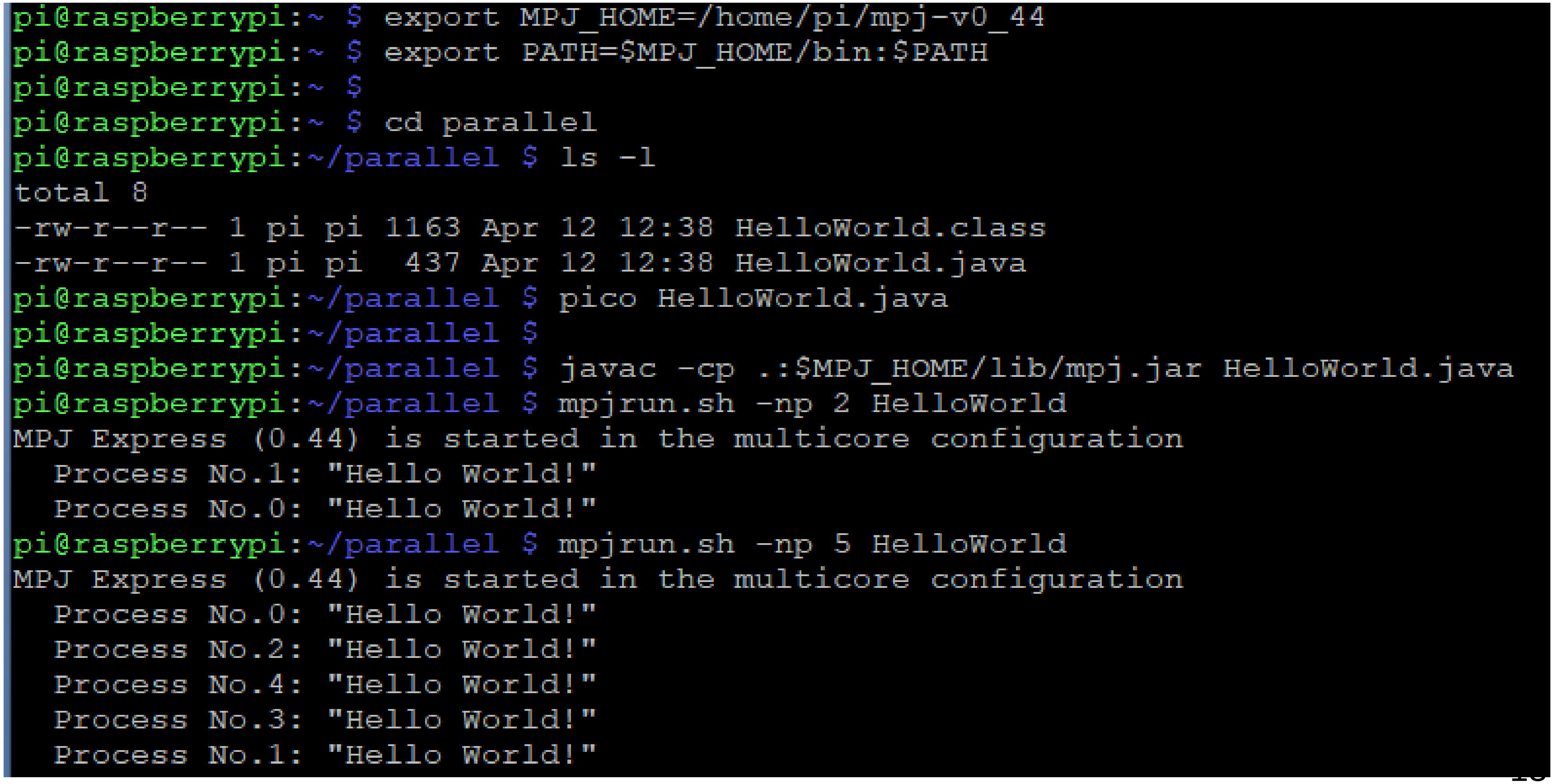
```
import mpi.*;
public class Parallel_1_Basic
{
    public static void main(String args[]) throws Exception
    {
      int rank = 0;
      int size = 0;

      MPI.Init(args);

      size = MPI.COMM_WORLD.Size();
      rank = MPI.COMM_WORLD.Rank();

      System.out.println(“Process No."+rank+": \"Hello World!\"");
      MPI.Finalize();
    }
}
```

# First Program

- Sample of Execution Results
  - In essence, each process executes autonomously.
  - The messages do not necessarily print in order
  - If five separate processes are running on different processors, and it cannot know beforehand which one will execute its print statement first.
  - If the processes are being scheduled on the same processor instead of multiple processors, then it is up to the operating system to schedule the processes, and it has no preference of any one of the processes over any other process of ours.



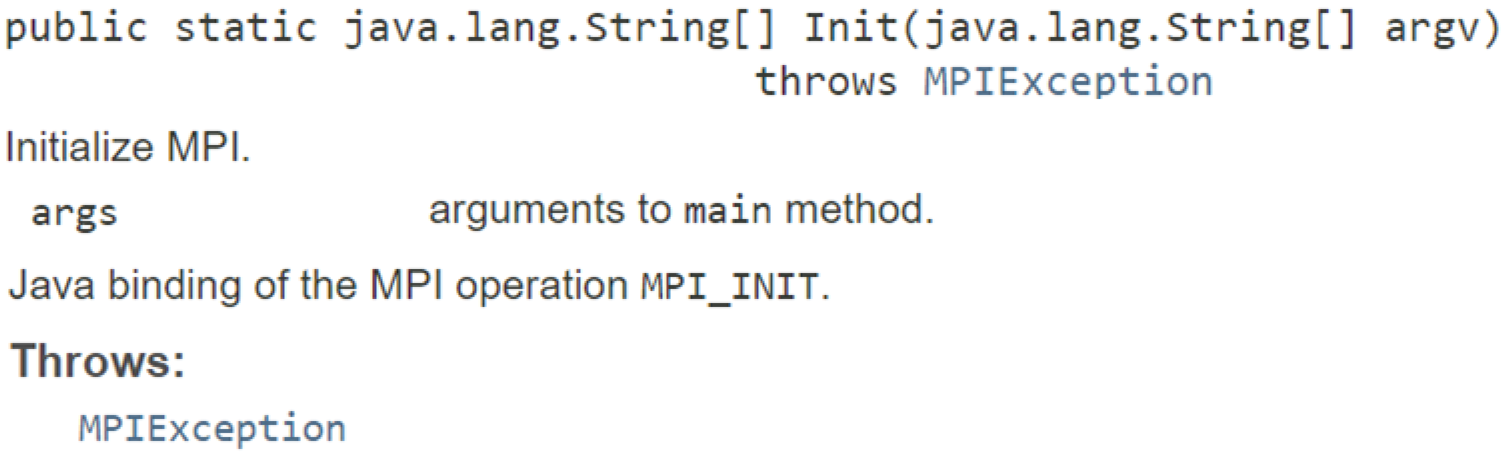
```
MPJ Express (0.44) is started in the multicore configuration
  Process No.1: "Hello World!"
  Process No.0: "Hello World!"
  Process No.2: "Hello World!"
  Process No.4: "Hello World!"
  Process No.3: "Hello World!"
```



```
pi@raspberrypi:~ $ export MPJ_HOME=/home/pi/mpj-v0_44
pi@raspberrypi:~ $ export PATH=$MPJ_HOME/bin:$PATH
pi@raspberrypi:~ $
pi@raspberrypi:~ $ cd parallel
pi@raspberrypi:~/parallel $ ls -l
total 8
-rw-r--r-- 1 pi pi 1163 Apr 12 12:38 HelloWorld.class
-rw-r--r-- 1 pi pi  437 Apr 12 12:38 HelloWorld.java
pi@raspberrypi:~/parallel $ pico HelloWorld.java
pi@raspberrypi:~/parallel $
pi@raspberrypi:~/parallel $ javac -cp .:$MPJ_HOME/lib/mpj.jar HelloWorld.java
pi@raspberrypi:~/parallel $ mpjrun.sh -np 2 HelloWorld
MPJ Express (0.44) is started in the multicore configuration
  Process No.1: "Hello World!"
  Process No.0: "Hello World!"
pi@raspberrypi:~/parallel $ mpjrun.sh -np 5 HelloWorld
MPJ Express (0.44) is started in the multicore configuration
  Process No.0: "Hello World!"
  Process No.2: "Hello World!"
  Process No.4: "Hello World!"
  Process No.3: "Hello World!"
  Process No.1: "Hello World!"
```

# First Program

- Explanation
  - import mpi.*
    - It needs to import the "*mpi*" package to make available the MPI
  - MPI.Init(args)



```
public static java.lang.String[] Init(java.lang.String[] argv)
                                  throws MPIException
```

Initialize MPI.

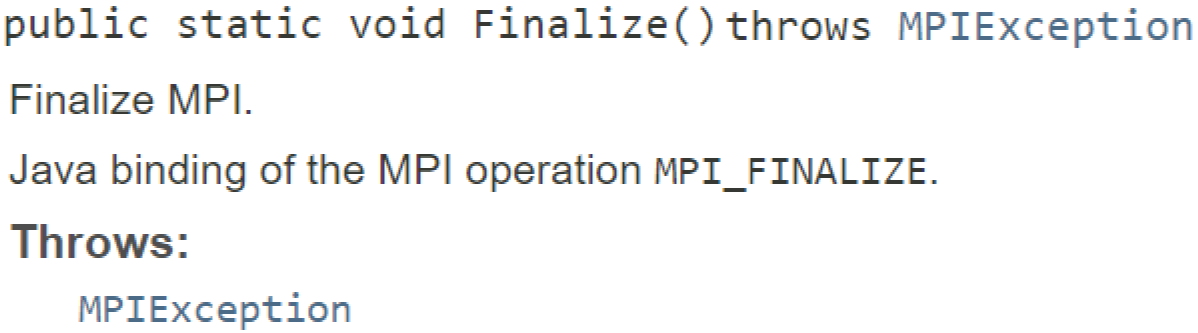
| | |
|---|---|
| `args` | arguments to `main` method. |

Java binding of the MPI operation `MPI_INIT`.

**Throws:**

`MPIException`

  - MPI.Finalize()



```
public static void Finalize() throws MPIException
```

Finalize MPI.

Java binding of the MPI operation `MPI_FINALIZE`.

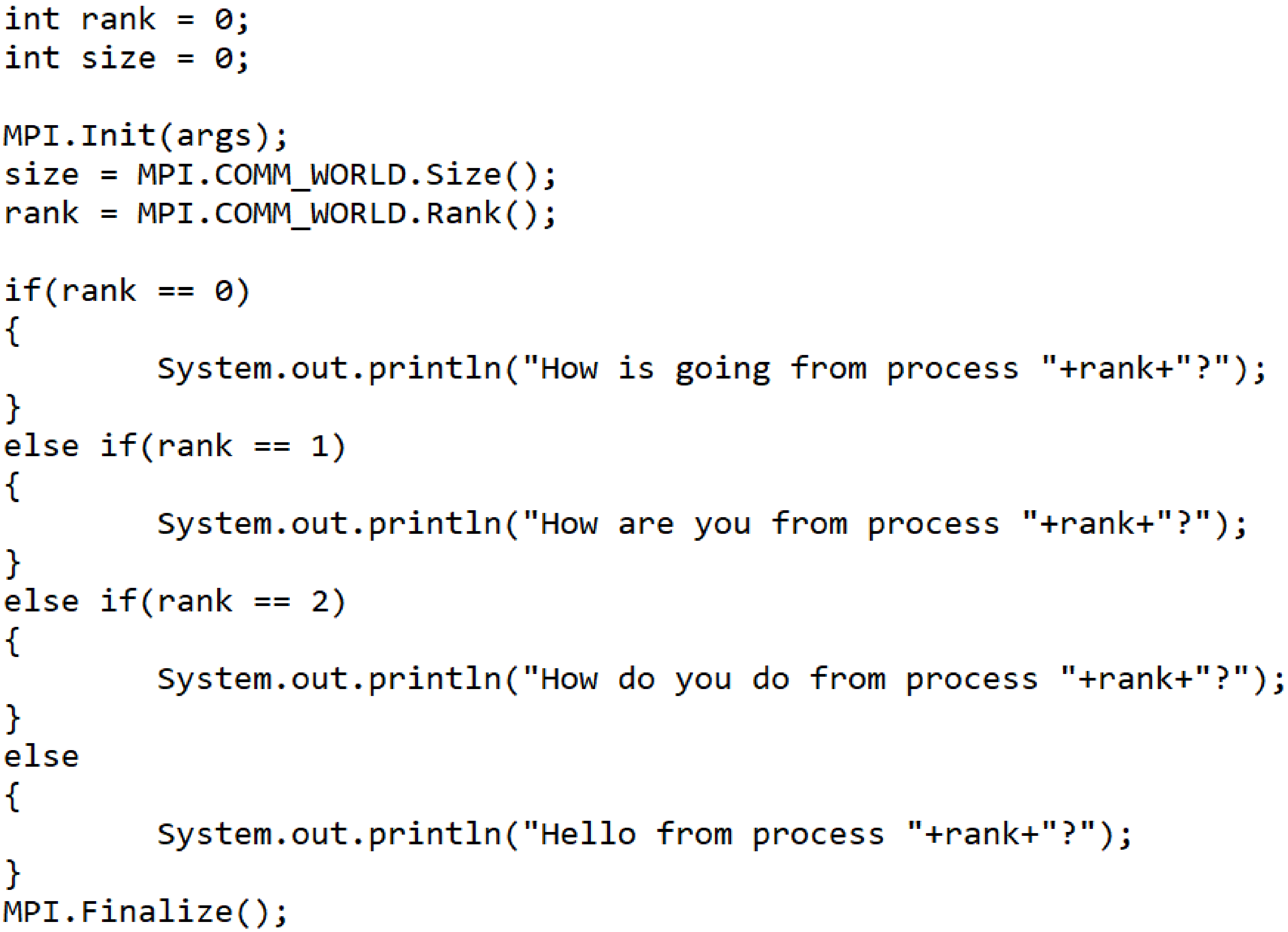
**Throws:**

`MPIException`

# First Program

- Explanation (continued)
  - *MPI.COMM_WORLD*
    - A predefined communicator
    - It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in the group of MPI_COMM_WORLD.
  - MPI.COMM_WORLD.Size()
    - Size() returns the number of processors in the group of this communicator
  - MPI.COMM_WORLD.Rank()
    - Rank() returns the rank of the calling process in the group of this communicator
- Separate codes in one file
  - When an MPI program runs, each process receives the same code.
  - However, each process can be assigned a different task.
    - This allows us to embed a separate code for each process into one file.

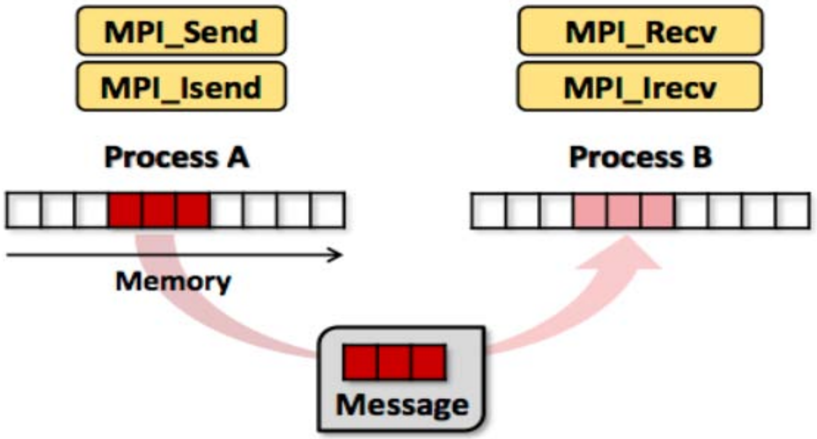
# First Program



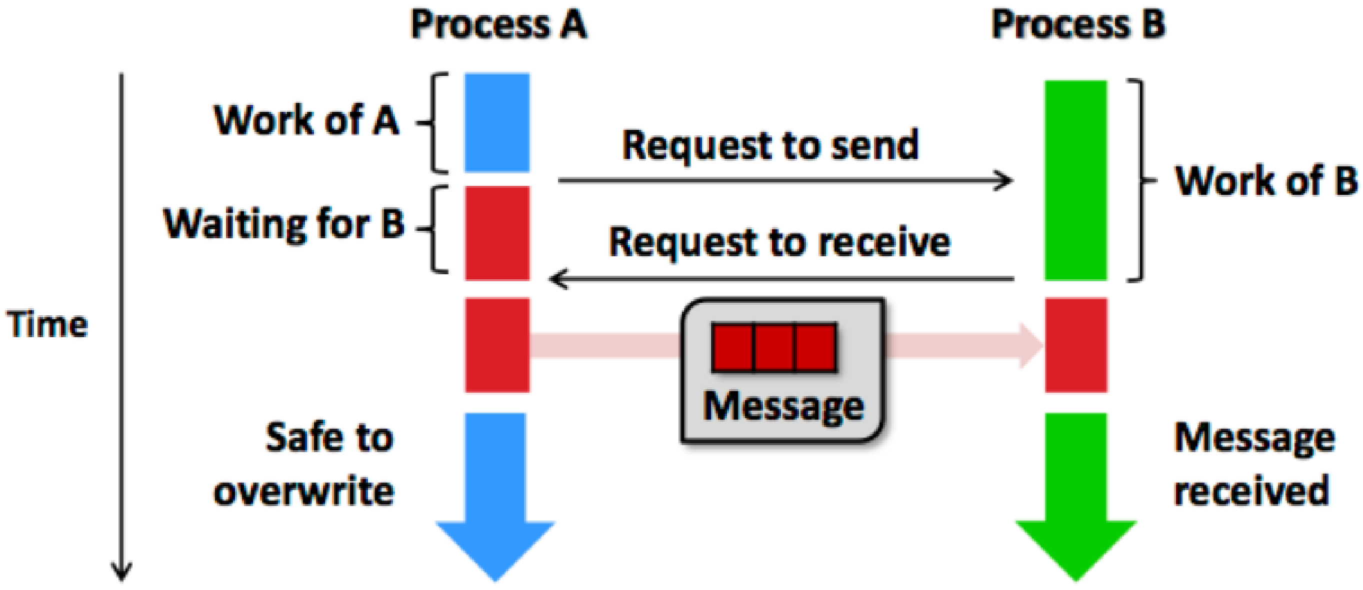
```java
int rank = 0;
int size = 0;

MPI.Init(args);
size = MPI.COMM_WORLD.Size();
rank = MPI.COMM_WORLD.Rank();

if(rank == 0)
{
        System.out.println("How is going from process "+rank+"?");
}
else if(rank == 1)
{
        System.out.println("How are you from process "+rank+"?");
}
else if(rank == 2)
{
        System.out.println("How do you do from process "+rank+"?");
}
else
{
        System.out.println("Hello from process "+rank+"?");
}
MPI.Finalize();
```

# Point-to-Point Communications and Collective Operations

- MPJ supports point-to-point communications and collective operations
  - Point-to-point communications
    - A communication between two processes: send and receive.
  - Collective operations
    - The communication that involves a group or groups of processes, e.g., broadcast, scatter, gather, and reduce.
- Blocking communications
  - A blocking send operation terminates when the message is received by the destination.
    - I.e., a program that invokes a blocking send operation will block until the message is received by the destination.
  - A blocking receive operation terminates when a message is received by the caller.
    - I.e., a program that invokes a blocking receive primitive will block until a message is received by the caller.
  - Example



**In a point-to-point communication, a piece of data (a message) is copied from the memory of one process to the memory of another process.**



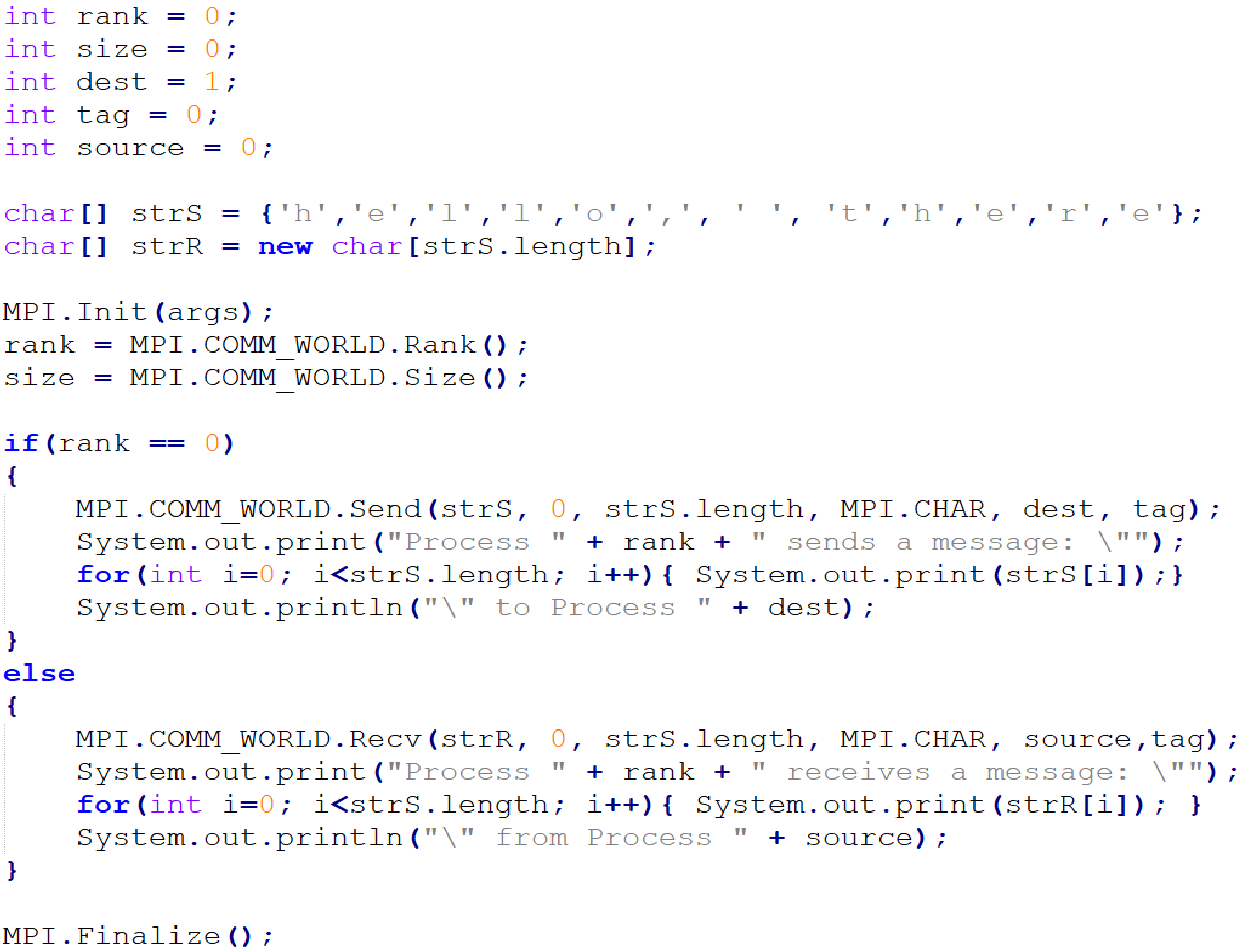
With a blocking send, no other operations can be executed until the communication has completed.

https://skirt.ugent.be/skirt9/_parallel_messaging.html

**Example 3.1** A simple ‘hello world’ example usage of point-to-point communication.

```
#include "mpi.h"
int main(int argc, char *argv[])
{
  char message[20];
  int myrank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0)    /* code for process zero */
  {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else if (myrank == 1)  /* code for process one */
  {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

In Example 3.1, process zero (`myrank = 0`) sends a *message* to process one using the *send* operation MPI_SEND. The operation specifies a *send buffer* in the sender memory from which the *message data* is taken. In the example above, the send buffer consists of the storage containing the variable `message` in the memory of process zero. The location, size and type of the send buffer are specified by the first three parameters of the send operation. The message sent will contain the 13 characters of this variable. In addition, the send operation associates an *envelope* with the message. This *envelope* specifies the message destination and contains distinguishing information that can be used by the *receive* operation to select a particular message. The last three parameters of the send operation, along with the rank of the sender, specify the *envelope* for the message sent. Process one (`myrank = 1`) receives this message with the *receive* operation MPI_RECV. The message to be received is selected according to the value of its *envelope*, and the *message data* is stored into the *receive buffer*. In the example above, the receive buffer consists of the storage containing the string `message` in the memory of process one. The first three parameters of the receive operation specify the location, size and type of the receive buffer. The next three parameters are used for selecting the incoming message. The last parameter is used to return information on the message just received.



```
int rank = 0;
int size = 0;
int dest = 1;
int tag = 0;
int source = 0;

char[] strS = {'h','e','l','l','o',',', ' ', 't','h','e','r','e'};
char[] strR = new char[strS.length];

MPI.Init(args);
rank = MPI.COMM_WORLD.Rank();
size = MPI.COMM_WORLD.Size();

if(rank == 0)
{
    MPI.COMM_WORLD.Send(strS, 0, strS.length, MPI.CHAR, dest, tag);
    System.out.print("Process " + rank + " sends a message: \"");
    for(int i=0; i<strS.length; i++){ System.out.print(strS[i]);}
    System.out.println("\" to Process " + dest);
}
else
{
    MPI.COMM_WORLD.Recv(strR, 0, strS.length, MPI.CHAR, source,tag);
    System.out.print("Process " + rank + " receives a message: \"");
    for(int i=0; i<strS.length; i++){ System.out.print(strR[i]); }
    System.out.println("\" from Process " + source);
}

MPI.Finalize();
```

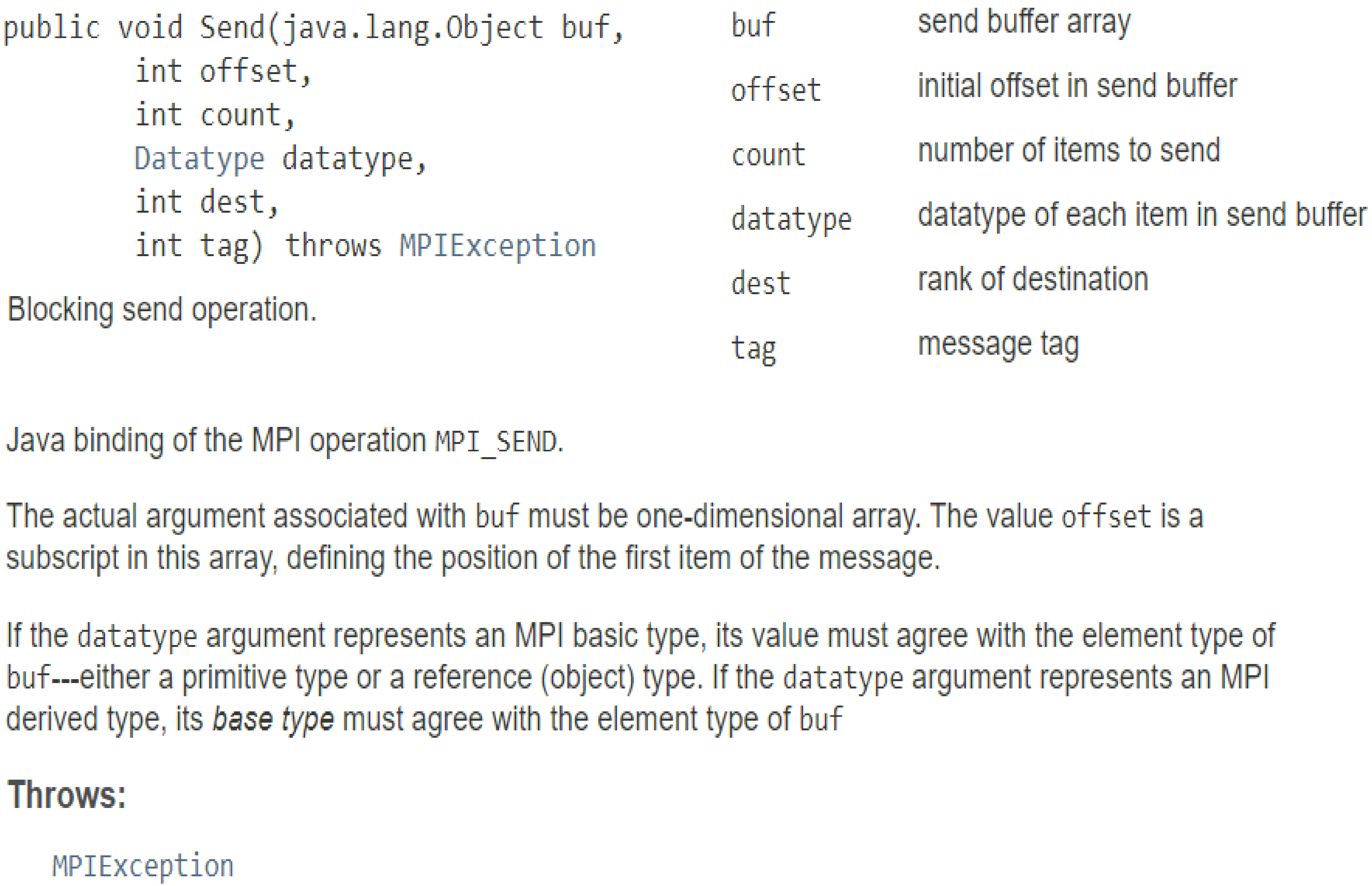
The syntax of the **blocking send** procedure is given below.

MPI_SEND(buf, count, datatype, dest, tag, comm)

| | | |
|---|---|---|
| IN | buf | initial address of send buffer (choice) |
| IN | count | number of elements in send buffer (non-negative integer) |
| IN | datatype | datatype of each send buffer element (handle) |
| IN | dest | rank of destination (integer) |
| IN | tag | message tag (integer) |
| IN | comm | communicator (handle) |

- IN: the call may use the input value but does not update the argument from the perspective of the caller at any time during the call's execution,
- OUT: the call may update the argument but does not use its input value,
- INOUT: the call may both use and update the argument.

# Point-to-Point Communication – Send()



```
public void Send(java.lang.Object buf,
        int offset,
        int count,
        Datatype datatype,
        int dest,
        int tag) throws MPIException
```

Blocking send operation.

| | |
|---|---|
| `buf` | send buffer array |
| `offset` | initial offset in send buffer |
| `count` | number of items to send |
| `datatype` | datatype of each item in send buffer |
| `dest` | rank of destination |
| `tag` | message tag |

Java binding of the MPI operation `MPI_SEND`.

The actual argument associated with `buf` must be one-dimensional array. The value `offset` is a subscript in this array, defining the position of the first item of the message.

If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`---either a primitive type or a reference (object) type. If the `datatype` argument represents an MPI derived type, its *base type* must agree with the element type of `buf`

**Throws:**

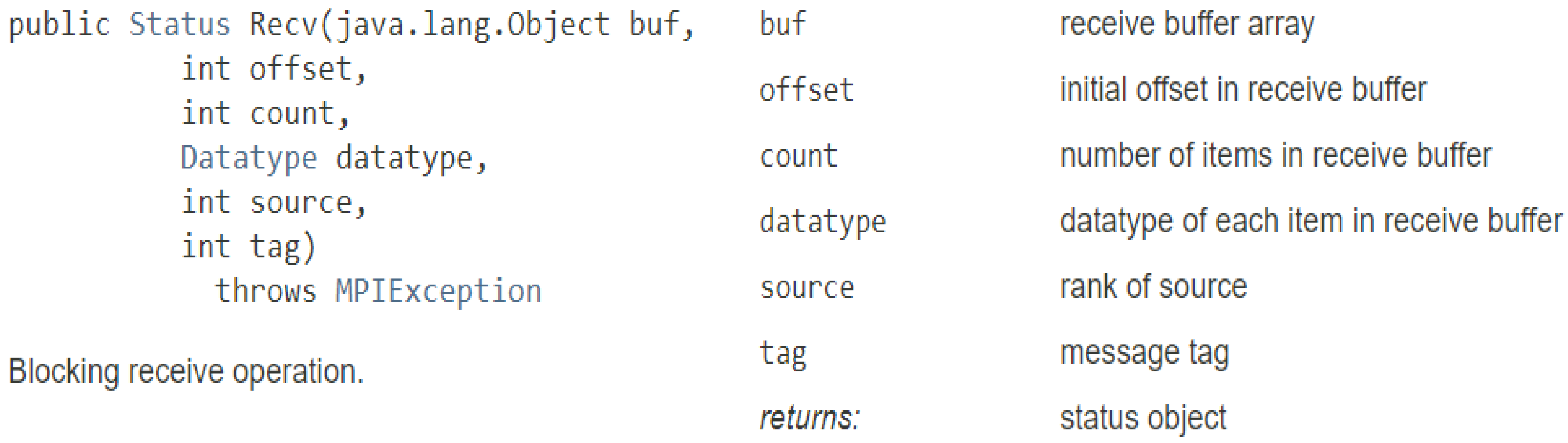
`MPIException`

The syntax of the **blocking receive** procedure is given below.

MPI_RECV(buf, count, datatype, source, tag, comm, status)

| | | |
|---|---|---|
| OUT | buf | initial address of receive buffer (choice) |
| IN | count | number of elements in receive buffer (non-negative integer) |
| IN | datatype | datatype of each receive buffer element (handle) |
| IN | source | rank of source or MPI_ANY_SOURCE (integer) |
| IN | tag | message tag or MPI_ANY_TAG (integer) |
| IN | comm | communicator (handle) |
| OUT | status | status object (status) |

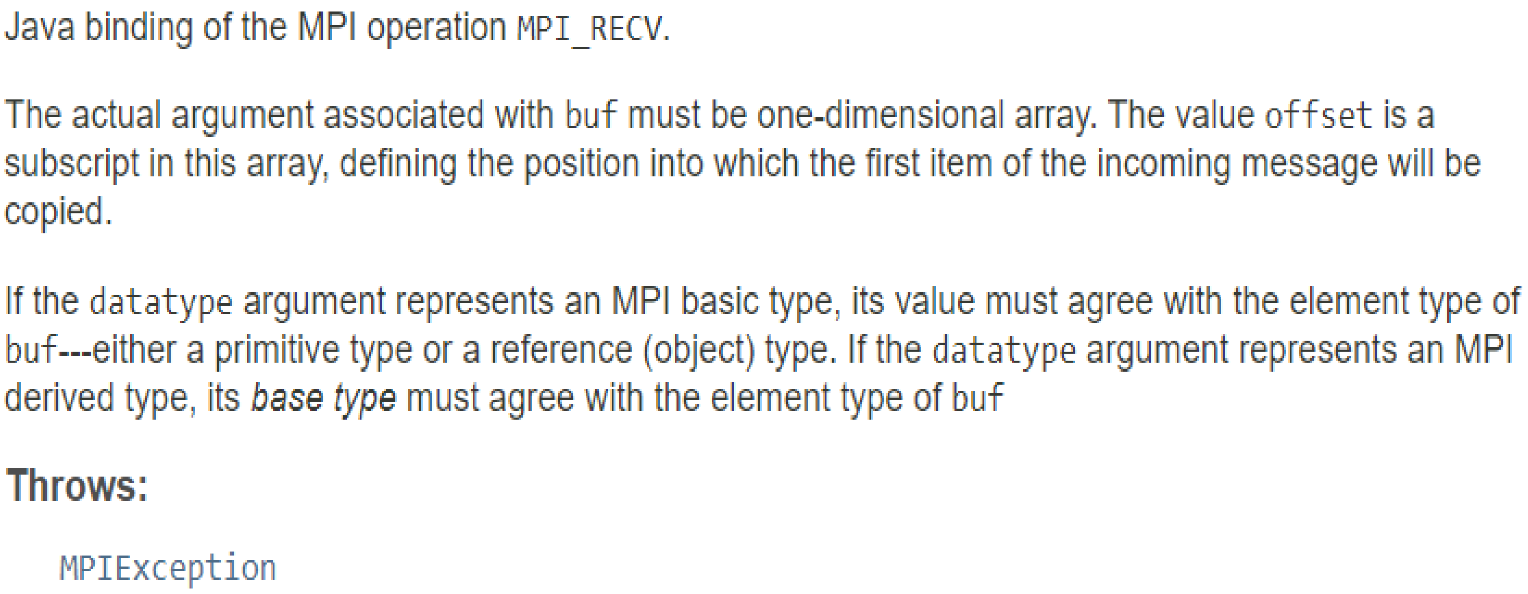
# Point-to-Point Communication – Recv()



```
public Status Recv(java.lang.Object buf,
          int offset,
          int count,
          Datatype datatype,
          int source,
          int tag)
            throws MPIException
```

Blocking receive operation.

| | |
|---|---|
| `buf` | receive buffer array |
| `offset` | initial offset in receive buffer |
| `count` | number of items in receive buffer |
| `datatype` | datatype of each item in receive buffer |
| `source` | rank of source |
| `tag` | message tag |
| *returns:* | status object |



Java binding of the MPI operation `MPI_RECV`.

The actual argument associated with `buf` must be one-dimensional array. The value `offset` is a subscript in this array, defining the position into which the first item of the incoming message will be copied.

If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`---either a primitive type or a reference (object) type. If the `datatype` argument represents an MPI derived type, its *base type* must agree with the element type of `buf`

**Throws:**

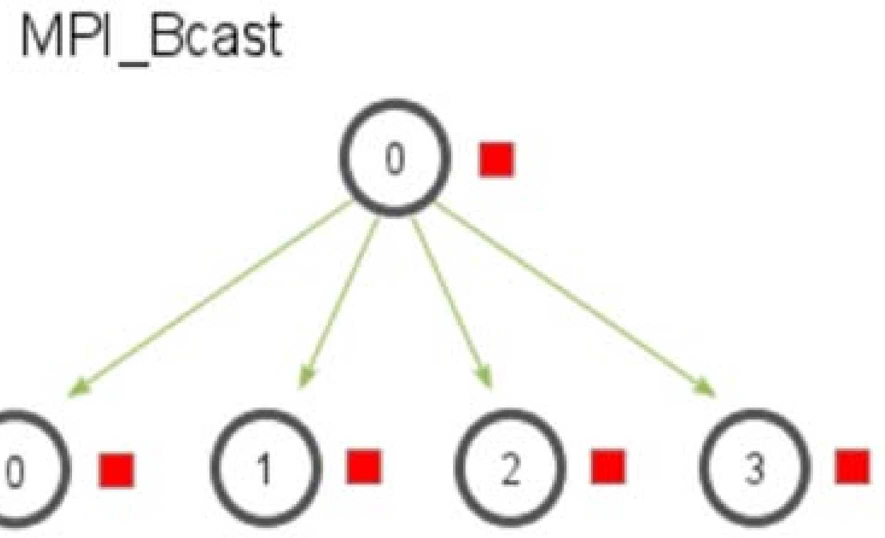
`MPIException`

# Collective Communication

- Collective communication is defined as communication that involves a group or groups of processes.
  - Broadcast
  - Gather
  - Scatter
  - All gather
  - Reduce
  - All reduce
  - All total

# Collective Operations - Broadcast

- Broadcast
  - Broadcast a message from one member process to all members of a group (including itself).

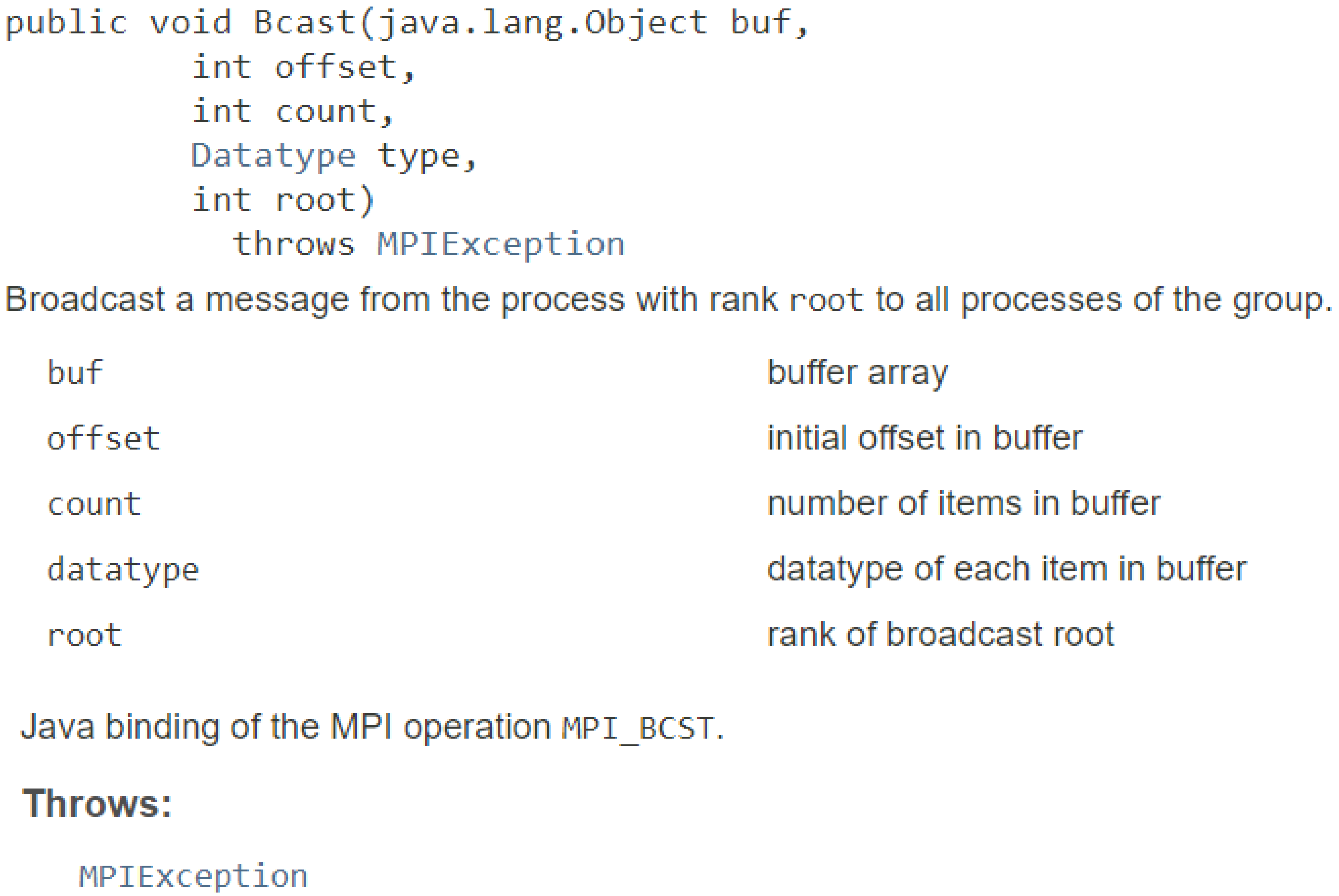


  - Example

MPI_BCAST(buffer, count, datatype, root, comm)

| | | |
|---|---|---|
| INOUT | buffer | starting address of buffer (choice) |
| IN | count | number of entries in buffer (non-negative integer) |
| IN | datatype | datatype of buffer (handle) |
| IN | root | rank of broadcast root (integer) |
| IN | comm | communicator (handle) |

# Collective Operations - Broadcast



```
public void Bcast(java.lang.Object buf,
         int offset,
         int count,
         Datatype type,
         int root)
           throws MPIException
```

Broadcast a message from the process with rank `root` to all processes of the group.

| | |
|---|---|
| `buf` | buffer array |
| `offset` | initial offset in buffer |
| `count` | number of items in buffer |
| `datatype` | datatype of each item in buffer |
| `root` | rank of broadcast root |

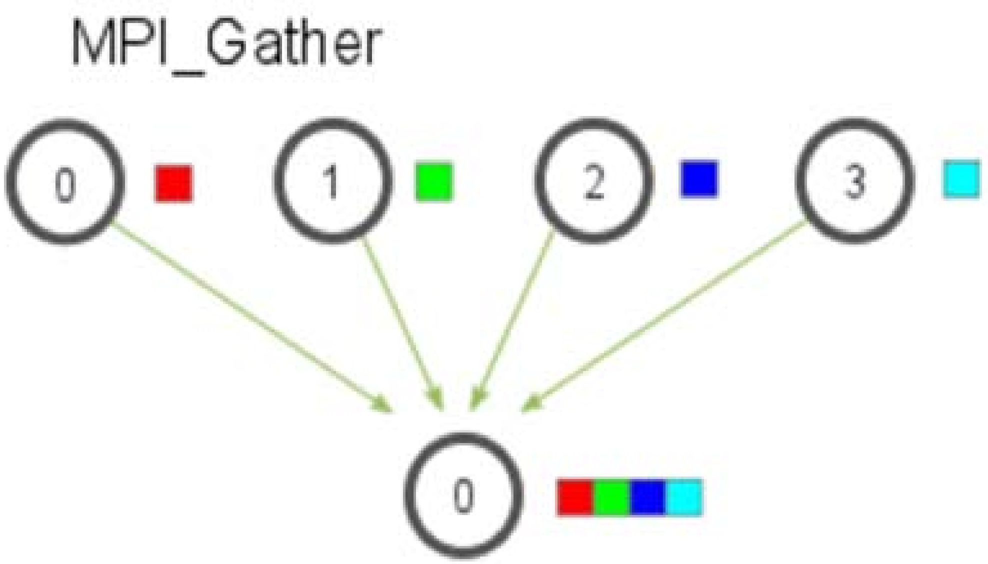
Java binding of the MPI operation `MPI_BCST`.

**Throws:**

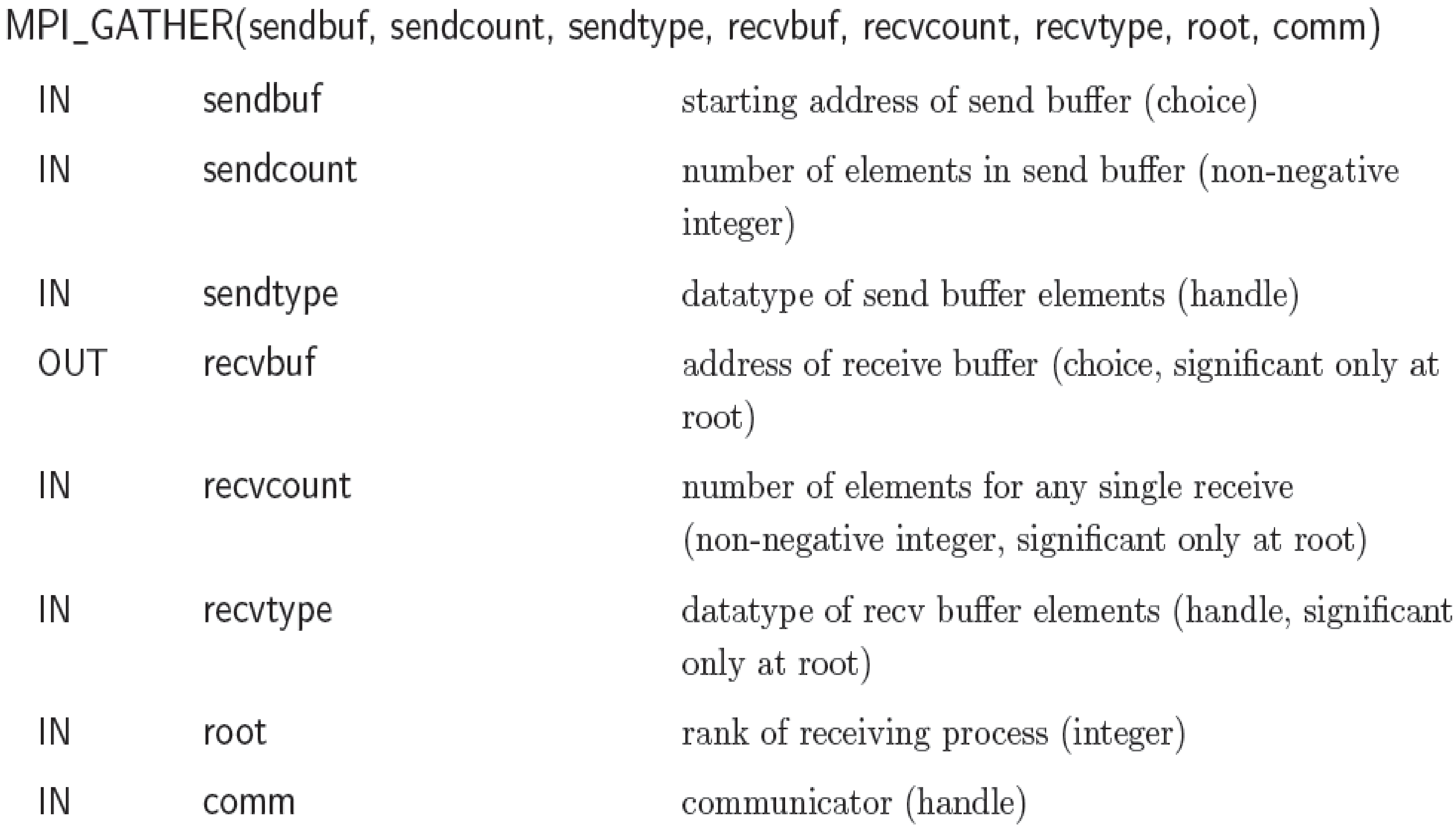
`MPIException`

# Collective Operations - Gather

- Gather
  - Gather data from all members of a group to one member.



  - Example

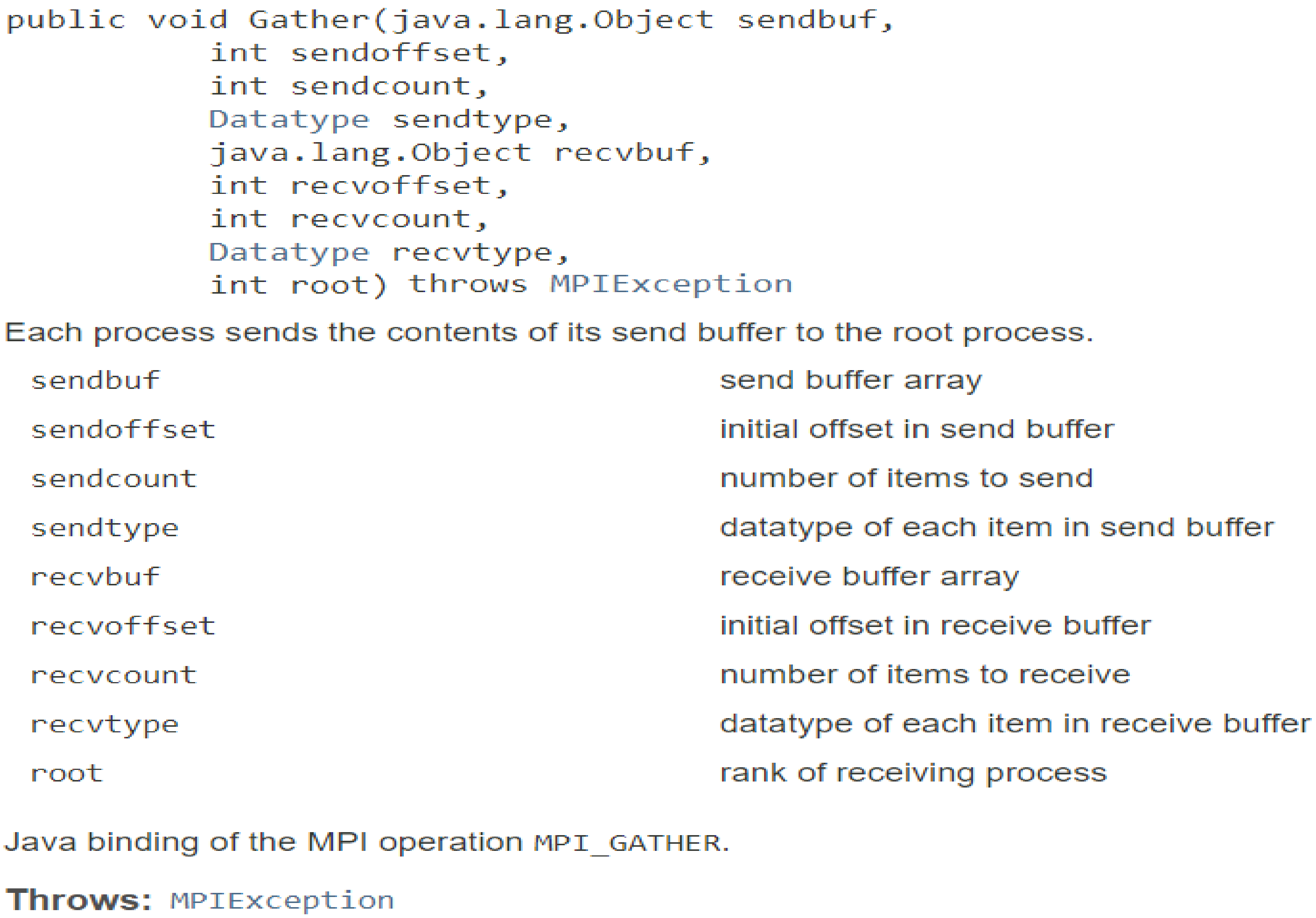


MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements in send buffer (non-negative integer) |
| IN | sendtype | datatype of send buffer elements (handle) |
| OUT | recvbuf | address of receive buffer (choice, significant only at root) |
| IN | recvcount | number of elements for any single receive (non-negative integer, significant only at root) |
| IN | recvtype | datatype of recv buffer elements (handle, significant only at root) |
| IN | root | rank of receiving process (integer) |
| IN | comm | communicator (handle) |

- The *n* messages sent by the processes in the group are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcountn, recvtype, ...).
- The receive buffer is ignored for all non-root processes.

# Collective Operations - Gather



```
public void Gather(java.lang.Object sendbuf,
          int sendoffset,
          int sendcount,
          Datatype sendtype,
          java.lang.Object recvbuf,
          int recvoffset,
          int recvcount,
          Datatype recvtype,
          int root) throws MPIException
```

Each process sends the contents of its send buffer to the root process.

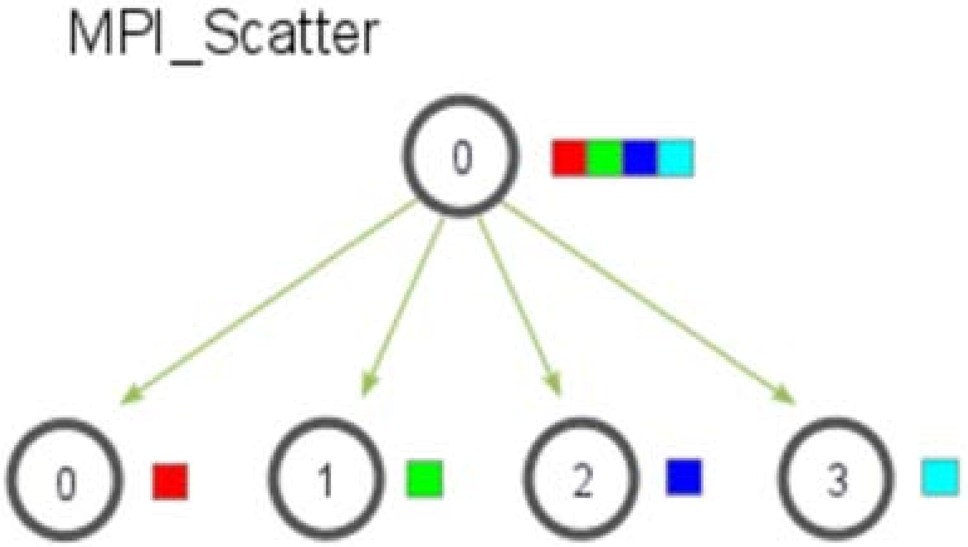
| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `sendcount` | number of items to send |
| `sendtype` | datatype of each item in send buffer |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `recvcount` | number of items to receive |
| `recvtype` | datatype of each item in receive buffer |
| `root` | rank of receiving process |

Java binding of the MPI operation `MPI_GATHER`.

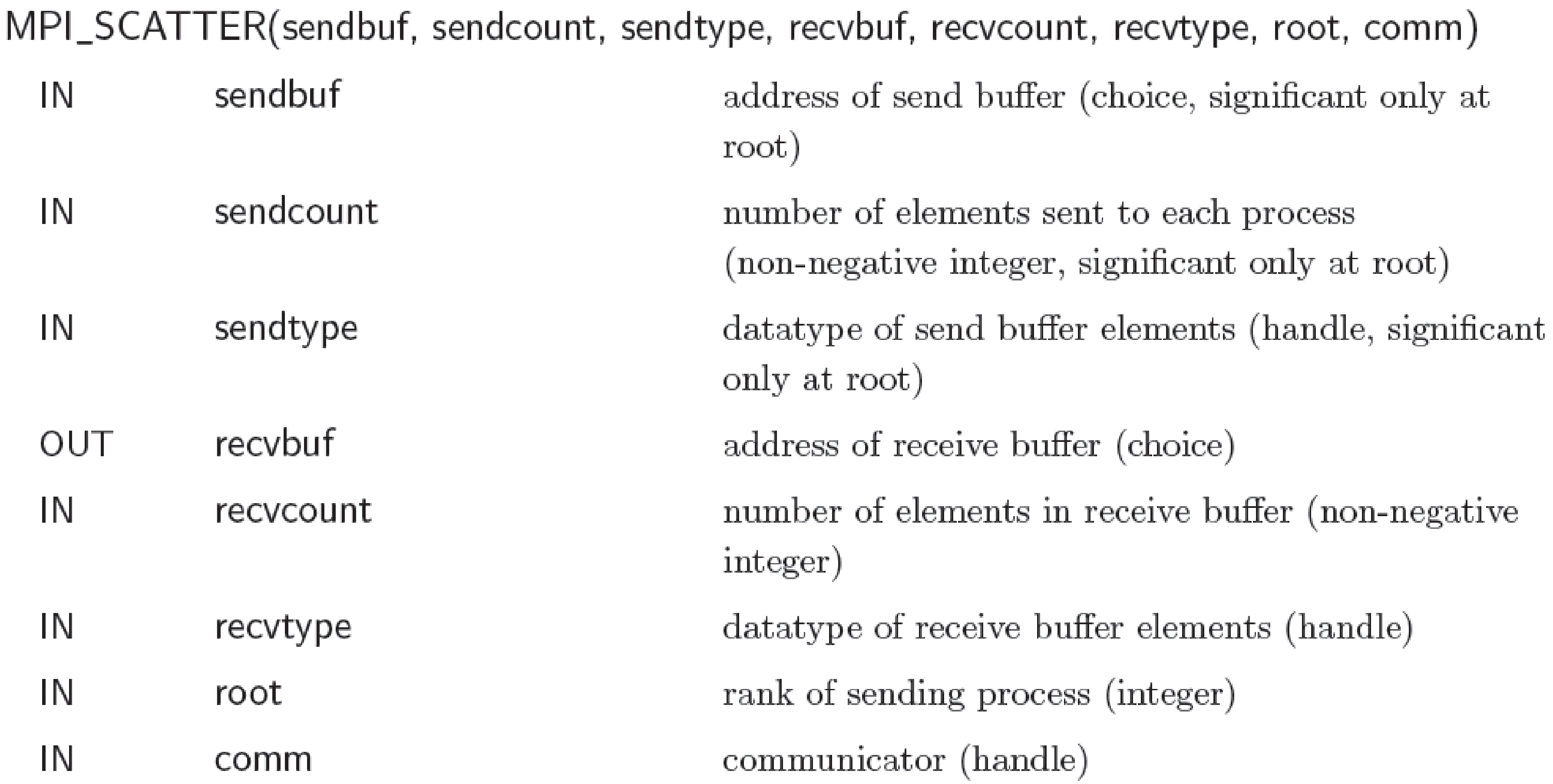
**Throws:** `MPIException`

# Collective Operations - Scatter

- Scatter
  - Split a data into N parts and send each part to each member process of a group.



  - Example

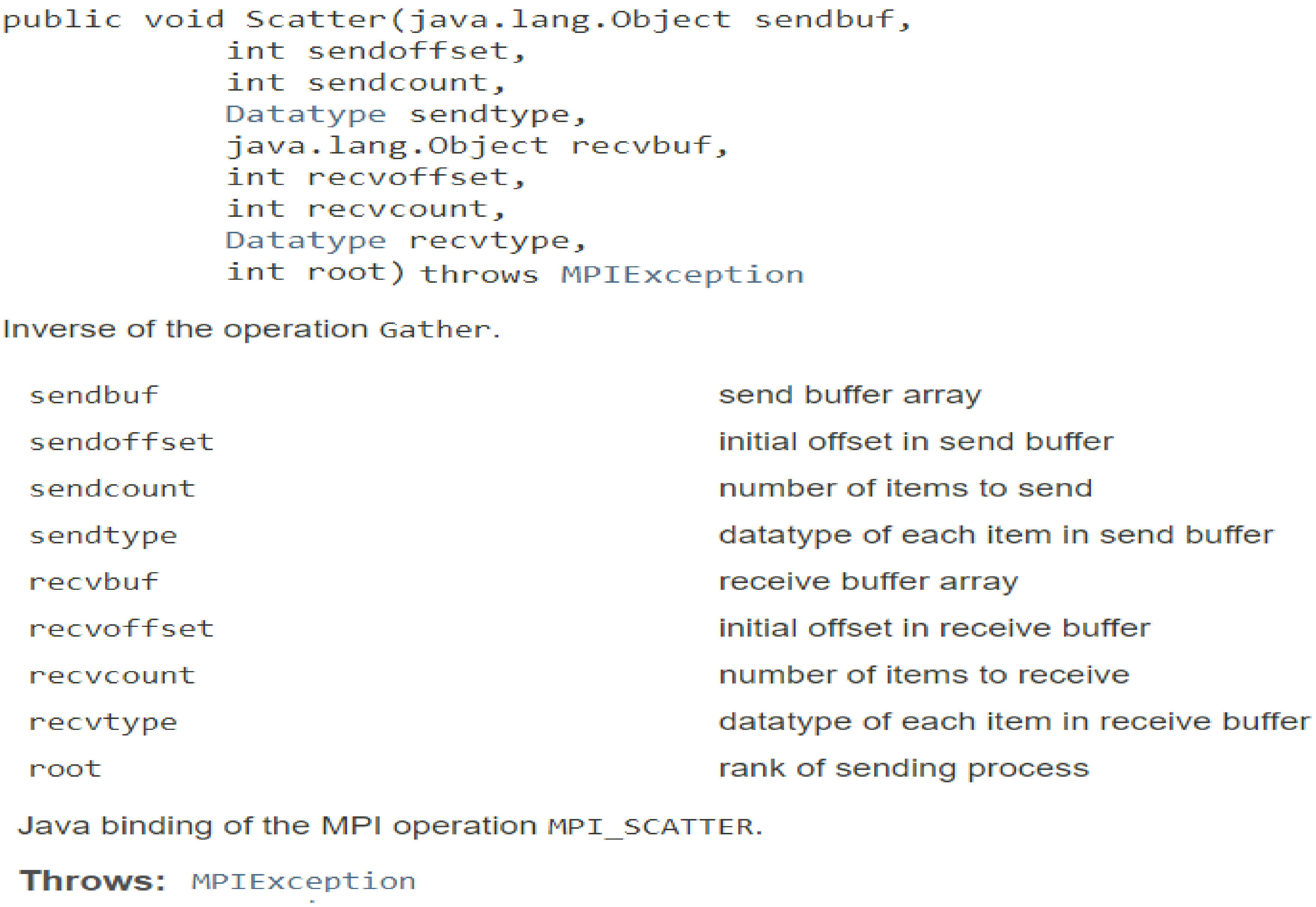


MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

| | | |
|---|---|---|
| IN | sendbuf | address of send buffer (choice, significant only at root) |
| IN | sendcount | number of elements sent to each process (non-negative integer, significant only at root) |
| IN | sendtype | datatype of send buffer elements (handle, significant only at root) |
| OUT | recvbuf | address of receive buffer (choice) |
| IN | recvcount | number of elements in receive buffer (non-negative integer) |
| IN | recvtype | datatype of receive buffer elements (handle) |
| IN | root | rank of sending process (integer) |
| IN | comm | communicator (handle) |

- The root sends a message with MPI_Send(sendbuf, sendcountn, sendtype, …).
- This message is split into n equal segments, the *i*th segment is sent to the *i*th process in the group, and each process receives this message as above.
- The send buffer is ignored for all non-root processes.

# Collective Operations - Scatter



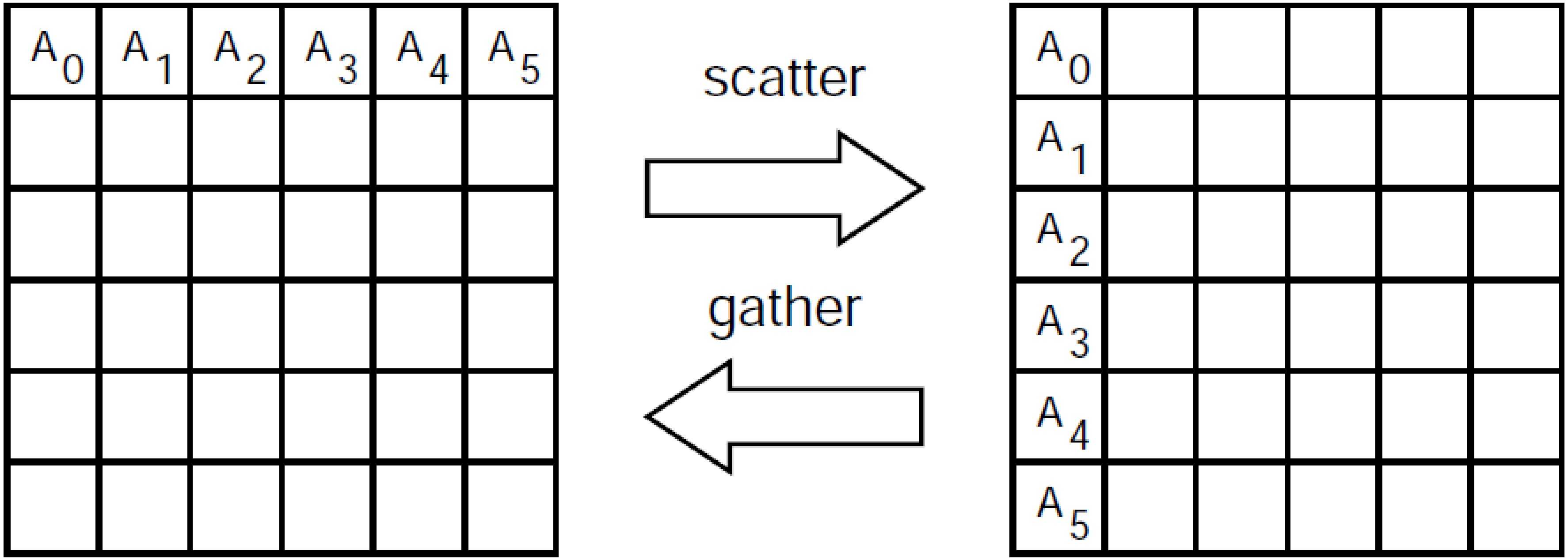
```
public void Scatter(java.lang.Object sendbuf,
                    int sendoffset,
                    int sendcount,
                    Datatype sendtype,
                    java.lang.Object recvbuf,
                    int recvoffset,
                    int recvcount,
                    Datatype recvtype,
                    int root) throws MPIException
```

Inverse of the operation `Gather`.

| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `sendcount` | number of items to send |
| `sendtype` | datatype of each item in send buffer |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `recvcount` | number of items to receive |
| `recvtype` | datatype of each item in receive buffer |
| `root` | rank of sending process |

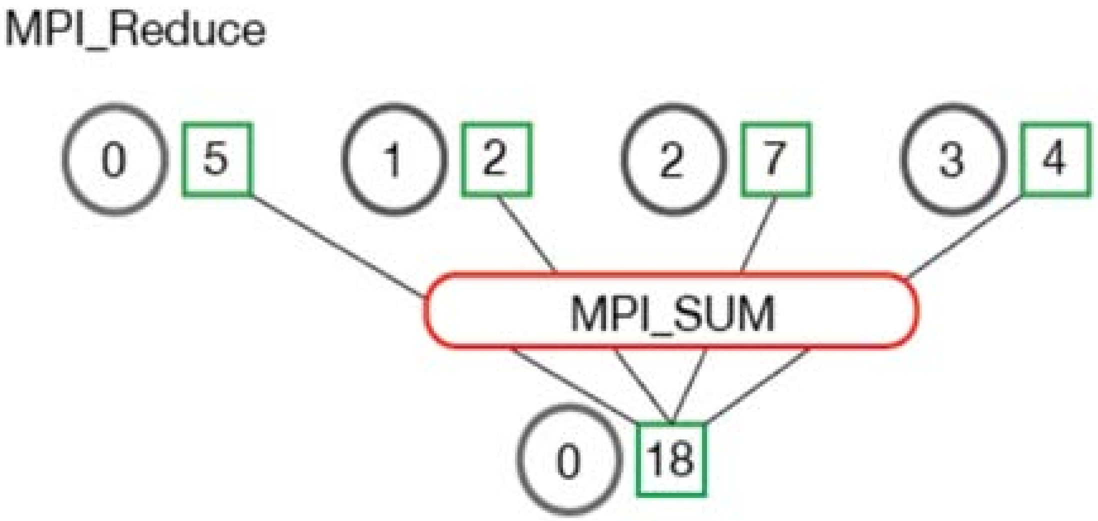
Java binding of the MPI operation `MPI_SCATTER`.

**Throws:** `MPIException`


$A_0$ $A_1$ $A_2$ $A_3$ $A_4$ $A_5$
scatter
gather
$A_0$
$A_1$
$A_2$
$A_3$
$A_4$
$A_5$

# Collective Operations - Reduce

- Reduce
  - Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to one member process.



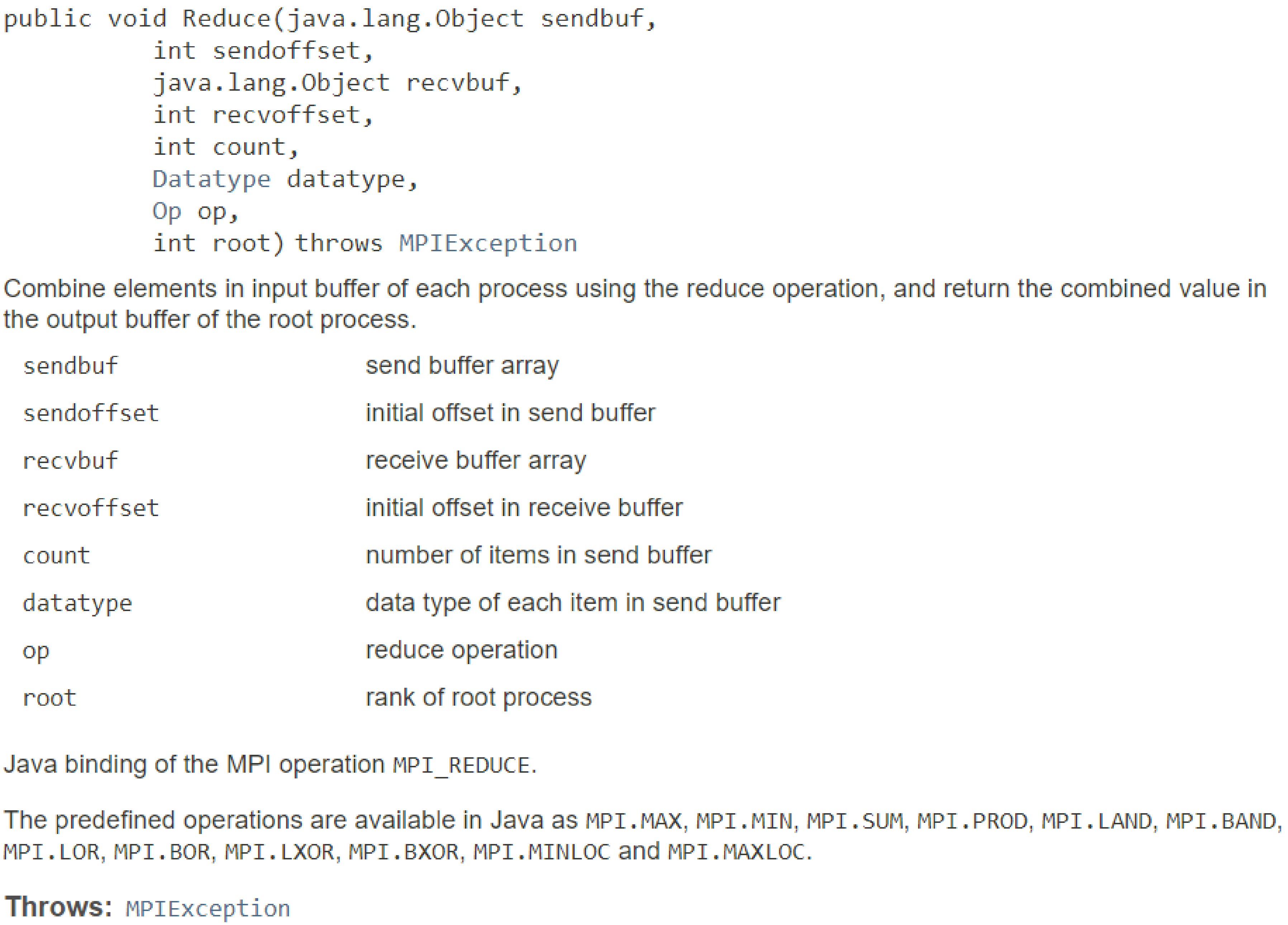
  - Example

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

| | | |
|---|---|---|
| IN | sendbuf | address of send buffer (choice) |
| OUT | recvbuf | address of receive buffer (choice, significant only at root) |
| IN | count | number of elements in send buffer (non-negative integer) |
| IN | datatype | datatype of elements of send buffer (handle) |
| IN | op | reduce operation (handle) |
| IN | root | rank of root process (integer) |
| IN | comm | communicator (handle) |

If comm is an intra-communicator, MPI_REDUCE combines the elements provided in the input buffer of each process in the group, using the operation op, and returns the combined value in the output buffer of the process with rank root. The input buffer is defined by the arguments sendbuf, count and datatype; the output buffer is defined by the arguments recvbuf, count and datatype; both have the same number of elements, with the same type. The routine is called by all group members using the same arguments for count, datatype, op, root and comm. Thus, all processes provide input buffers of the same length, with elements of the same type as the output buffer at the root. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is MPI_MAX and the send buffer contains two elements that are floating point numbers (count = 2 and datatype = MPI_FLOAT), then recvbuf(1) = *global* max(sendbuf(1)) and recvbuf(2) = *global* max(sendbuf(2)).

# Collective Operations - Reduce



```
public void Reduce(java.lang.Object sendbuf,
          int sendoffset,
          java.lang.Object recvbuf,
          int recvoffset,
          int count,
          Datatype datatype,
          Op op,
          int root) throws MPIException
```

Combine elements in input buffer of each process using the reduce operation, and return the combined value in the output buffer of the root process.

| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `count` | number of items in send buffer |
| `datatype` | data type of each item in send buffer |
| `op` | reduce operation |
| `root` | rank of root process |

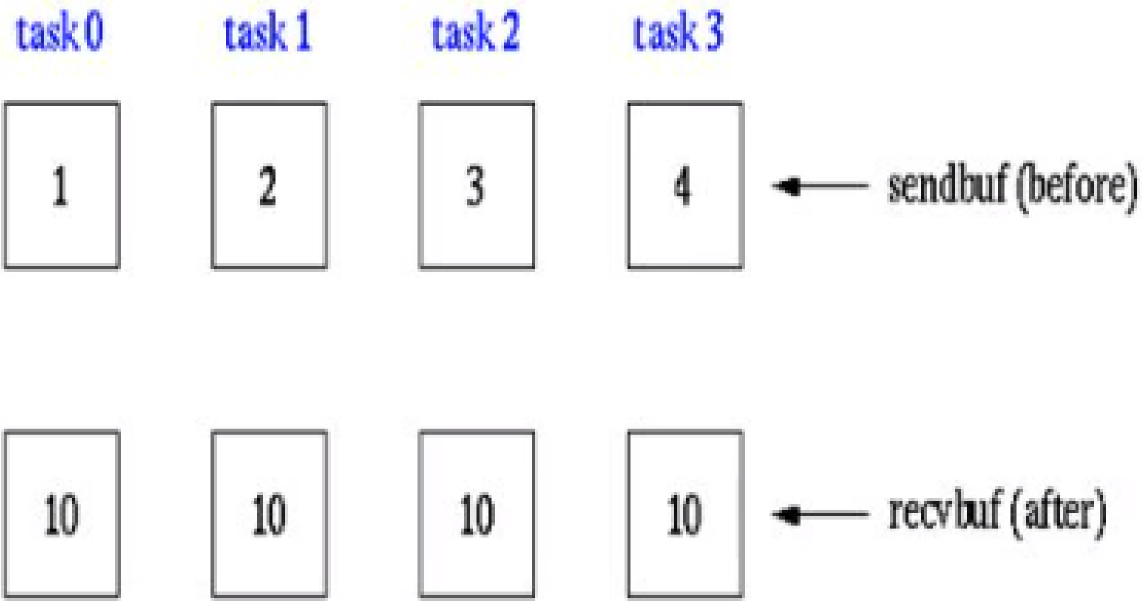
Java binding of the MPI operation `MPI_REDUCE`.

The predefined operations are available in Java as `MPI.MAX`, `MPI.MIN`, `MPI.SUM`, `MPI.PROD`, `MPI.LAND`, `MPI.BAND`, `MPI.LOR`, `MPI.BOR`, `MPI.LXOR`, `MPI.BXOR`, `MPI.MINLOC` and `MPI.MAXLOC`.

**Throws:** `MPIException`

# Collective Operations – All_Reduce

- All Reduce
  - The result is returned to all processes in a group.
  - All processes from the same group participating in these operations receive identical results.



  - Example

## MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| OUT | recvbuf | starting address of receive buffer (choice) |
| IN | count | number of elements in send buffer (non-negative integer) |
| IN | datatype | datatype of elements of send buffer (handle) |
| IN | op | operation (handle) |
| IN | comm | communicator (handle) |

```
public void Allreduce(java.lang.Object sendbuf,
                      int sendoffset,
                      java.lang.Object recvbuf,
                      int recvoffset,
                      int count,
                      Datatype datatype,
                      Op op)
                        throws MPIException
```

Same as `reduce` except that the result appears in receive buffer of all process in the group.

| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `count` | number of items in send buffer |
| `datatype` | data type of each item in send buffer |
| `op` | reduce operation |

Java binding of the MPI operation `MPI_ALLREDUCE`.

**Throws:**

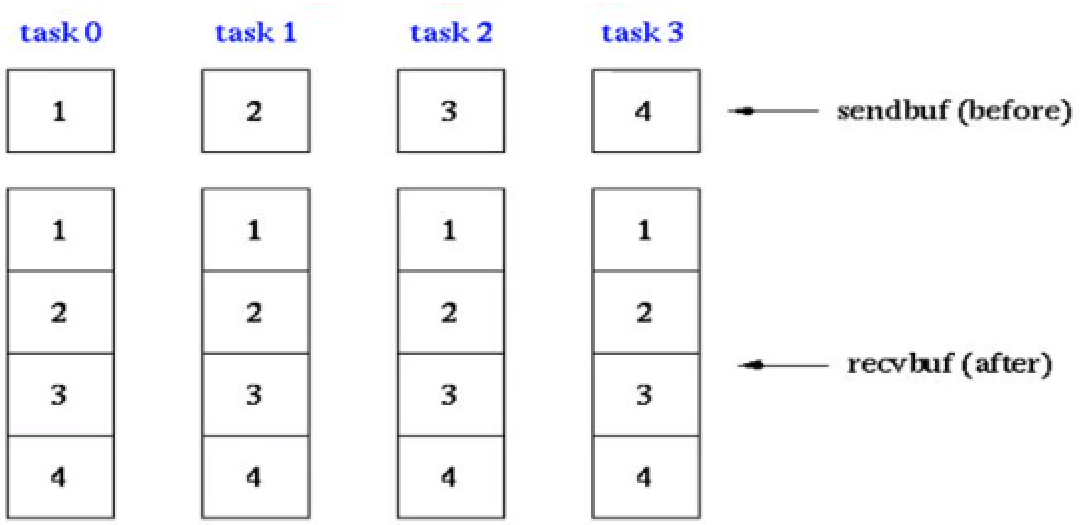
`MPIException`

# Collective Operations – All_Gather

- All_Gather
  - The outcome of a call to MPI_ALLGATHER(...) is as if all processes executed *n* calls to

    MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

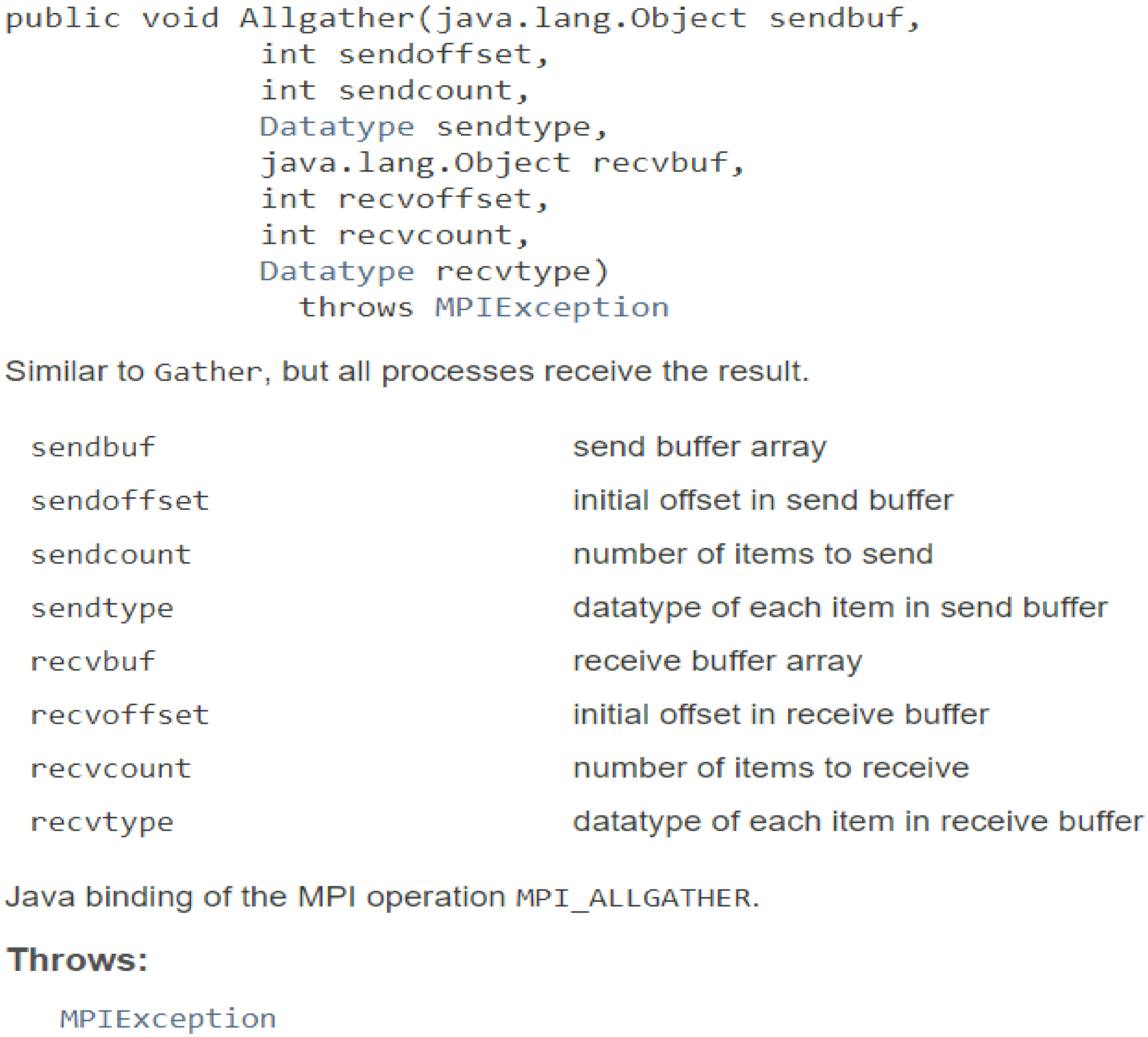
    where root = 0, …, n-1



  - Example

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements in send buffer (non-negative integer) |
| IN | sendtype | datatype of send buffer elements (handle) |
| OUT | recvbuf | address of receive buffer (choice) |
| IN | recvcount | number of elements received from any process (non-negative integer) |
| IN | recvtype | datatype of receive buffer elements (handle) |
| IN | comm | communicator (handle) |



```
public void Allgather(java.lang.Object sendbuf,
            int sendoffset,
            int sendcount,
            Datatype sendtype,
            java.lang.Object recvbuf,
            int recvoffset,
            int recvcount,
            Datatype recvtype)
              throws MPIException
```

Similar to `Gather`, but all processes receive the result.

| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `sendcount` | number of items to send |
| `sendtype` | datatype of each item in send buffer |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `recvcount` | number of items to receive |
| `recvtype` | datatype of each item in receive buffer |

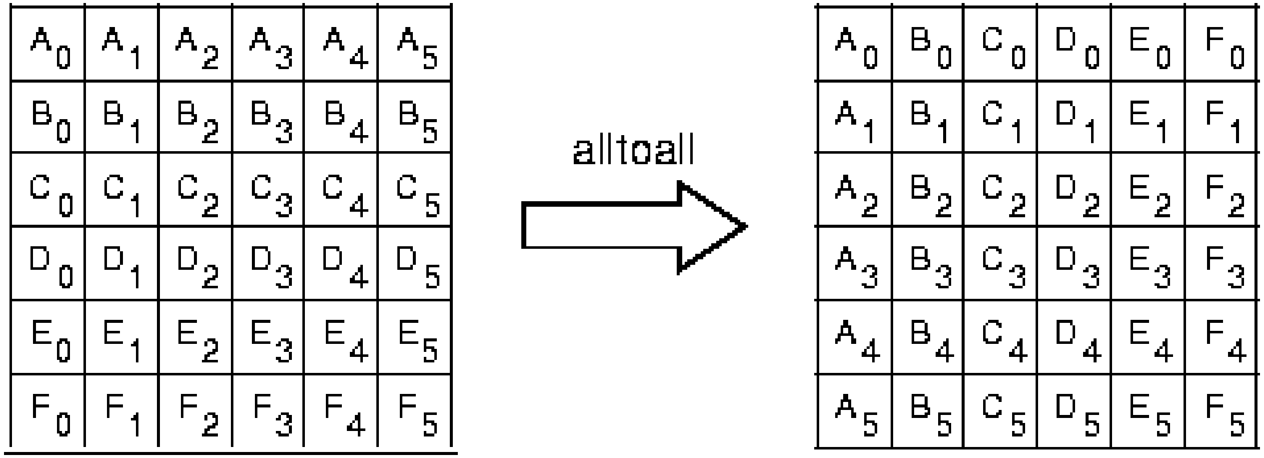
Java binding of the MPI operation `MPI_ALLGATHER`.

**Throws:**

`MPIException`

# Collective Operations – All_to_ALL

- All_to_ALL
  - Each process sends distinct data to each of the receivers.

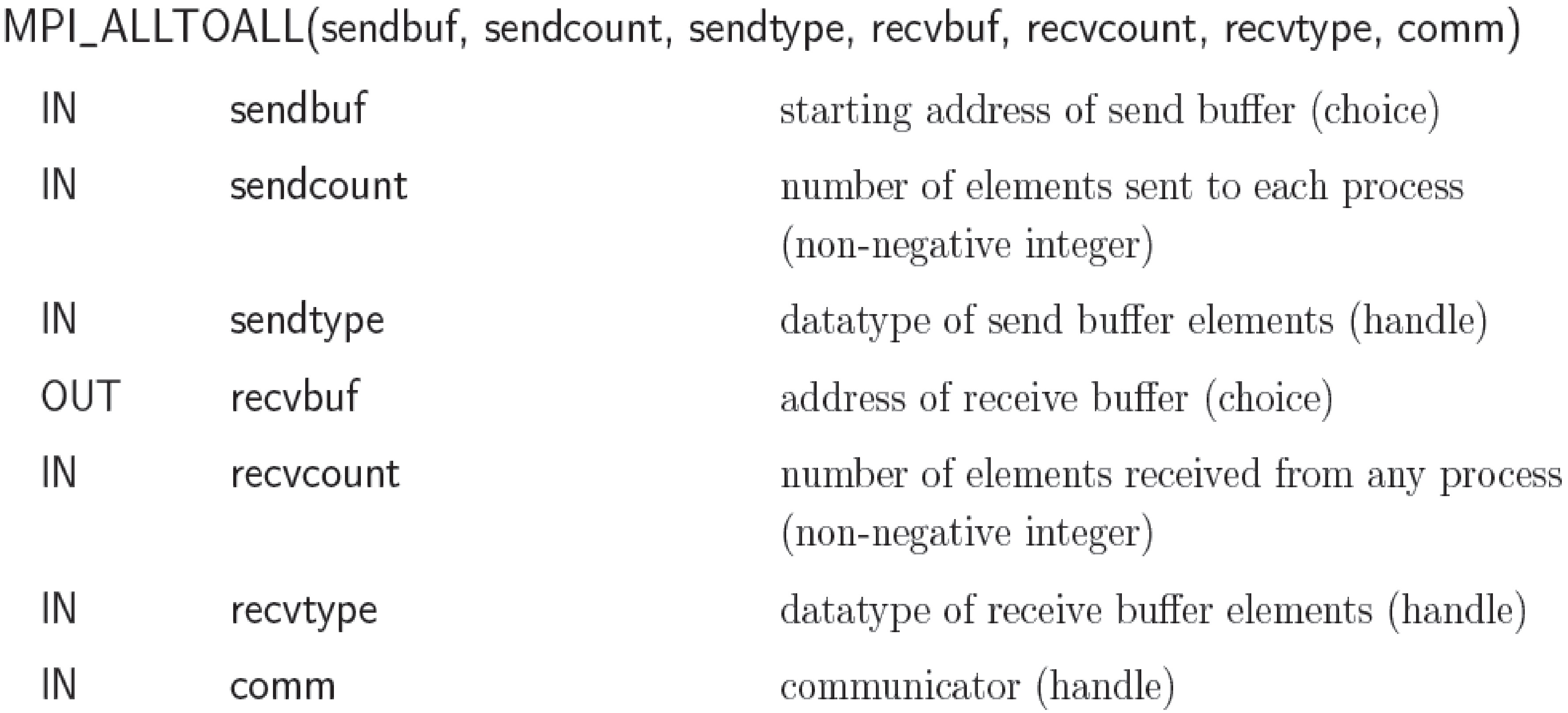


| | | | | | |
|---|---|---|---|---|---|
| $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| $B_0$ | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ |
| $C_0$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ |
| $E_0$ | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ |
| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ |

| | | | | | |
|---|---|---|---|---|---|
| $A_0$ | $B_0$ | $C_0$ | $D_0$ | $E_0$ | $F_0$ |
| $A_1$ | $B_1$ | $C_1$ | $D_1$ | $E_1$ | $F_1$ |
| $A_2$ | $B_2$ | $C_2$ | $D_2$ | $E_2$ | $F_2$ |
| $A_3$ | $B_3$ | $C_3$ | $D_3$ | $E_3$ | $F_3$ |
| $A_4$ | $B_4$ | $C_4$ | $D_4$ | $E_4$ | $F_4$ |
| $A_5$ | $B_5$ | $C_5$ | $D_5$ | $E_5$ | $F_5$ |

  - Example

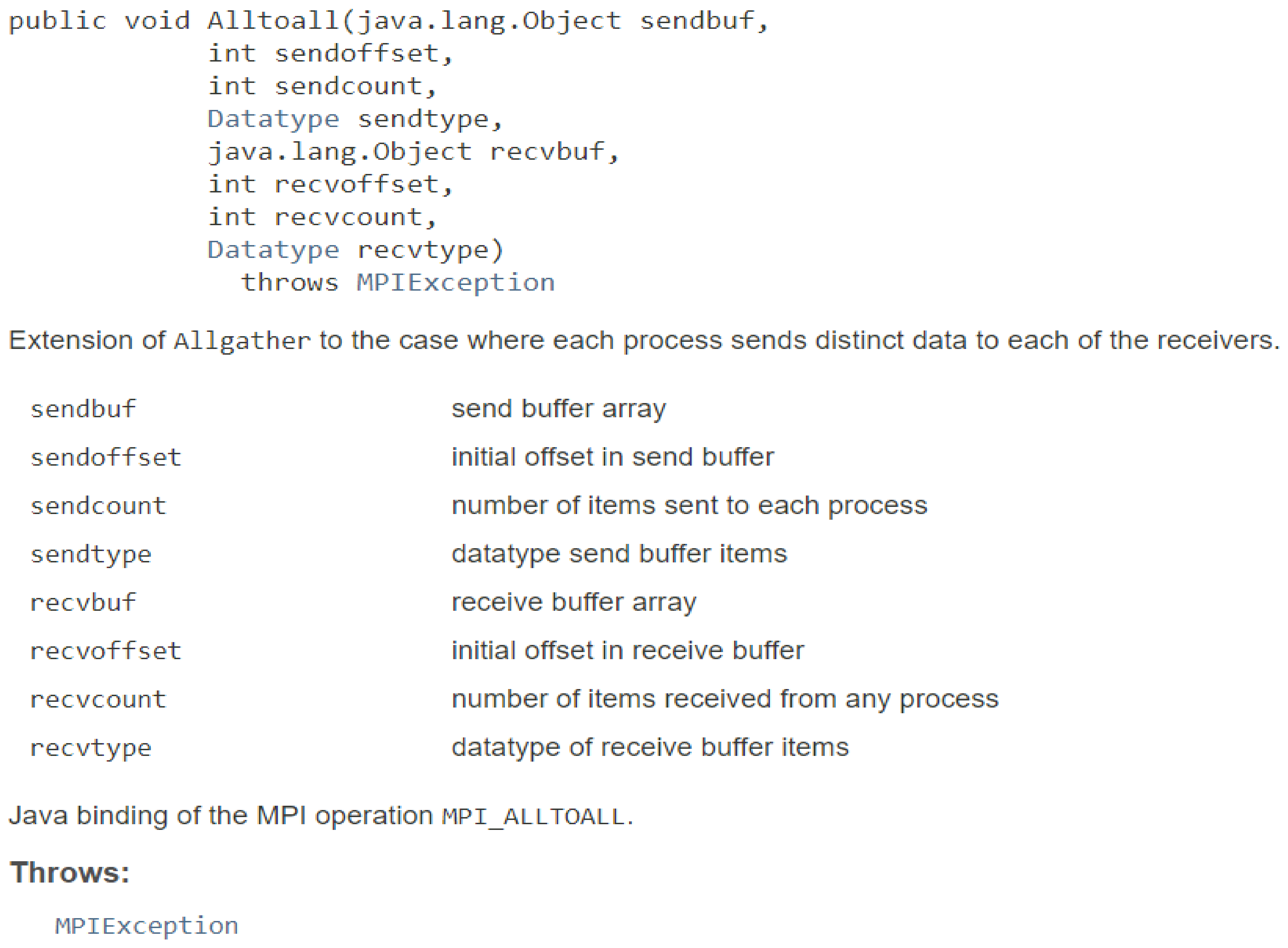
# Collective Operations – All_to_ALL



MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

| | | |
|---|---|---|
| IN | sendbuf | starting address of send buffer (choice) |
| IN | sendcount | number of elements sent to each process (non-negative integer) |
| IN | sendtype | datatype of send buffer elements (handle) |
| OUT | recvbuf | address of receive buffer (choice) |
| IN | recvcount | number of elements received from any process (non-negative integer) |
| IN | recvtype | datatype of receive buffer elements (handle) |
| IN | comm | communicator (handle) |

- Each process sends distinct data to each of the receivers.
- The *j*th block sent from process *i* is received by process *j* and is placed in the *i*th block of recvbuf.



```
public void Alltoall(java.lang.Object sendbuf,
            int sendoffset,
            int sendcount,
            Datatype sendtype,
            java.lang.Object recvbuf,
            int recvoffset,
            int recvcount,
            Datatype recvtype)
              throws MPIException
```

Extension of `Allgather` to the case where each process sends distinct data to each of the receivers.

| | |
|---|---|
| `sendbuf` | send buffer array |
| `sendoffset` | initial offset in send buffer |
| `sendcount` | number of items sent to each process |
| `sendtype` | datatype send buffer items |
| `recvbuf` | receive buffer array |
| `recvoffset` | initial offset in receive buffer |
| `recvcount` | number of items received from any process |
| `recvtype` | datatype of receive buffer items |

Java binding of the MPI operation `MPI_ALLTOALL`.

**Throws:**

`MPIException`

# References

- MPI 4.0 Standard
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- MPJ Express
  - http://mpjexpress.org/
- MPJ Express Documents
  - http://mpjexpress.org/docs/javadocs/index.html
- MPJ Express User Guide
  - Linux: http://mpjexpress.org/docs/guides/linuxguide.pdf
  - Windows: http://mpjexpress.org/docs/guides/windowsguide.pdf
- MPICH
  - https://www.mpich.org/
- Mpi4py
  - https://mpi4py.readthedocs.io/en/stable/

# References

- DeinoMPI
  - http://mpi.deino.net/
- Microsoft MPI
  - https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi