# MPI

http://www.mcs.anl.gov/research/projects/mpi/www/www3/

# Overview

1. Send/Recv
2. Collective Communication
3. Asynchronous Communication

# Send/Recv

# MPI Hello World

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

# MPI_Init

MPI_Init sets up the MPI runtime. The arguments to MPI_Init are argc and argv. In this case since we're taking void as the arguments to main we can pass in NULLs.

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

As a general rule, call MPI_Init first.

# MPI_Init

If we had command line arguments, MPI_Init would look like this. Note the changes to the main function.

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

const int MAX_STRING = 100;

int main(int argc, char** argv) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(argc, argv);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

# MPI_Finalize

MPI_Finalize lets the MPI runtime know that the program has been finished, and will deallocate the resources allocated for the MPI process(es).

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

# Communicators

MPI_COMM_WORLD is a *communicator* which refers to all the processes started by the user when they ran an MPI program.

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

For simplicity, we'll mostly be dealing with MPI_COMM_WORLD, but it is also possible to have more advanced groups of processes. This can be very useful for specific broadcast, scatter and gather operations.

# MPI_Comm_Size

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

MPI_Comm_Size will set the second argument (comm_sz) to the number of processes in the specified communicator.

This gives us the number of processes started by mpirun/ miexec.

# MPI_Comm_Rank

When you run MPI, you specify the number of processes -- each of these is a separate instance of the program being run (usually) on a different processor or core. They each have their own rank within the communicator.

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

MPI_Comm_Rank will set the second argument (my_rank) to the actual processes rank within the communicator.

`

```
int MPI_Send(
void*               msg_buf_p       /* in  */,
int                 msg_size        /* in  */,
MPI_Datatype        msg_type        /* in  */,
int                 dest            /* in  */,
int                 tag             /* in  */,
MPI_Comm            communicator    /* in  */);

int MPI_Recv(
void*               msg_buf_p       /* out */,
int                 buf_size        /* in  */,
MPI_Datatype        buf_type        /* in  */,
int                 source          /* in  */,
MPI_Comm            communicator    /* in  */,
MPI_Status*         status_p        /* out */);
```

An MPI_Send call needs to be paired to an MPI_Recv call.

msg_buf_p is a pointer to the message buffer to be sent, msg_size is it's size, and msg_type is the type of data in the buffer (the array type).

MPI_Comm_Rank will set the second argument (my_rank) to the actual processes rank within the communicator.

# Predefined MPI Datatypes

| Predefined MPI Datatypes | |
|---|---|
| **MPI datatype** | **C datatype** |
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_LONG_LONG | signed long long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | byte |
| MPI_PACKED | |

Note that you need to specify the data type for MPI_Send and MPI_Recv. Here are most of the predefined MPI datatypes.

# Message Matching

There are a few conditions for a message sent with MPI_Send to be received by MPI_Recv, these are:

```
recv_comm = send_comm
recv_tag = send_tag
dest = r
src = q
```

Where process q calls send:

```
MPI_Send(send_buff, send_buff_sz, send_type, dest, send_tag, send_comm);
```

And the process r calls recv:

```
MPI_Recv(recv_buff, recv_buff_sz, recv_type, src, recv_tag recv_comm, &status_p);
```

# Message Matching

Further, (depending on implementation) but generally:

```
recv_buff needs to have enough memory to hold send_buff
send_type = recv_type
recv_buff_sz >= send_buff_sz
```

# Recv Wildcards

It is also possible to receive from any process with a given tag with the `MPI_ANY_SOURCE` argument:

```
//On the master process
for (int i = 1; i < communicator_size; i++) {
      MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE, result_tag,
                comm, MPI_STATUS_IGNORE);
}
```

It is also possible to receive a message with any tag from a process with the `MPI_ANY_TAG` argument:

```
//On the master process
for (int i = 1; i < communicator_size; i++) {
      MPI_Recv(result, result_sz, result_type, i, MPI_ANY_TAG,
                comm, MPI_STATUS_IGNORE);
}
```

# Recv Wildcard

Or from any process and any tag:

```
//On the master process
for (int i = 1; i < communicator_size; i++) {
      MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE, MPI_ANY_TAG,
               comm, MPI_STATUS_IGNORE);
}
```

These allow you to hand situations where multiple messages could be coming in at the same time and you don't want to block waiting on Recvs to a process which hasn't finished yet, or on messages of a type you haven't received yet.

Note that the tag essentially lets you "name" a type of message with an id. This can be extremely useful in sorting out what messages are being sent and received.

# MPI_Send and MPI_Recv

Note that for every process that sends a message to process 0 using MPI_Send, there is a matching MPI_Recv call on process 0.

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

# A better MPI Hello World

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

const int MAX_STRING = 100;

int main(void) {
    char    greeting[MAX_STRING];
    int     comm_sz;    /* number of processes  */
    int     my_rank;    /* process rank         */

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank != 0) {
        sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
            printf("%s\n", greeting);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

We know that we will receive comm_sz - 1 messages, all of the same type. We can recv from any source (instead of from 1 .. comm_sz); which will let us receive them in the order sent, as opposed to blocking to receive them in order.

# status_p

Note that it's possible to receive a message without exactly knowing:

1. The source — MPI_ANY_SOURCE
2. The tag — MPI_ANY_TAG
3. The size of the buffer — recv_buff_sz >= send_buff_sz

The MPI_STATUS argument (the last one) in MPI_Recv lets us get this data.

# status_p

## MPI_Status is a struct with 3 elements:

```
MPI_SOURCE
MPI_TAG
MPI_ERROR
```

## So you can access them as follows:

```
//On the master process
for (int i = 1; i < communicator_size; i++) {
       MPI_Status status;
       MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE, MPI_ANY_TAG,
                comm, &status);

       cout << "Received a message with tag " << status.MPI_TAG
            << " from process " << status.MPI_SOURCE << endl;
}
```

# status_p

You can get the amount of data received with MPI_Get_count:

```
//On the master process
for (int i = 1; i < communicator_size; i++) {
       MPI_Status status;
       int count;
       MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE, MPI_ANY_TAG,
                comm, &status);
       MPI_Get_count(&status, result_type, &count);

       cout << "Received a message with tag " << status.MPI_TAG
            << " from process " << status.MPI_SOURCE
            << " with " << count << " elements." << endl;
}
```

This needs its own function because the datatype needs to be known.

# Semantics of Send & Recv

MPI_Send can have two behaviors, and they are dependent on the MPI implementation; so the same MPI code will not necessarily work the same way on two different systems.

Send can either *buffer*, where the contents of the message are placed into storage (to be send later when the paired up Recv call happens) and returns immediately allowing the program to continue without waiting on the message to be completed.

It can also *block*, where it waits for the paired Recv message to complete before returning.

# Semantics of Send & Recv

Buffering is generally better than blocking, as it lets the program proceed past the send without having to wait for the other process.

Typically, implementations will buffer sends of less than a certain size (because they have a limited buffer size), and block when the buffer is full.

MPI_Recv always *blocks*, until it has received the complete message.

# Potential Problems

Because a send can either block or buffer, its possible that given the underlying hardware and implementation your code can hang (or not run as fast as you expect).

Also if you don’t match up sends with receives, or have receives without a matching send your program will block indefinitely.

# Collective Communication

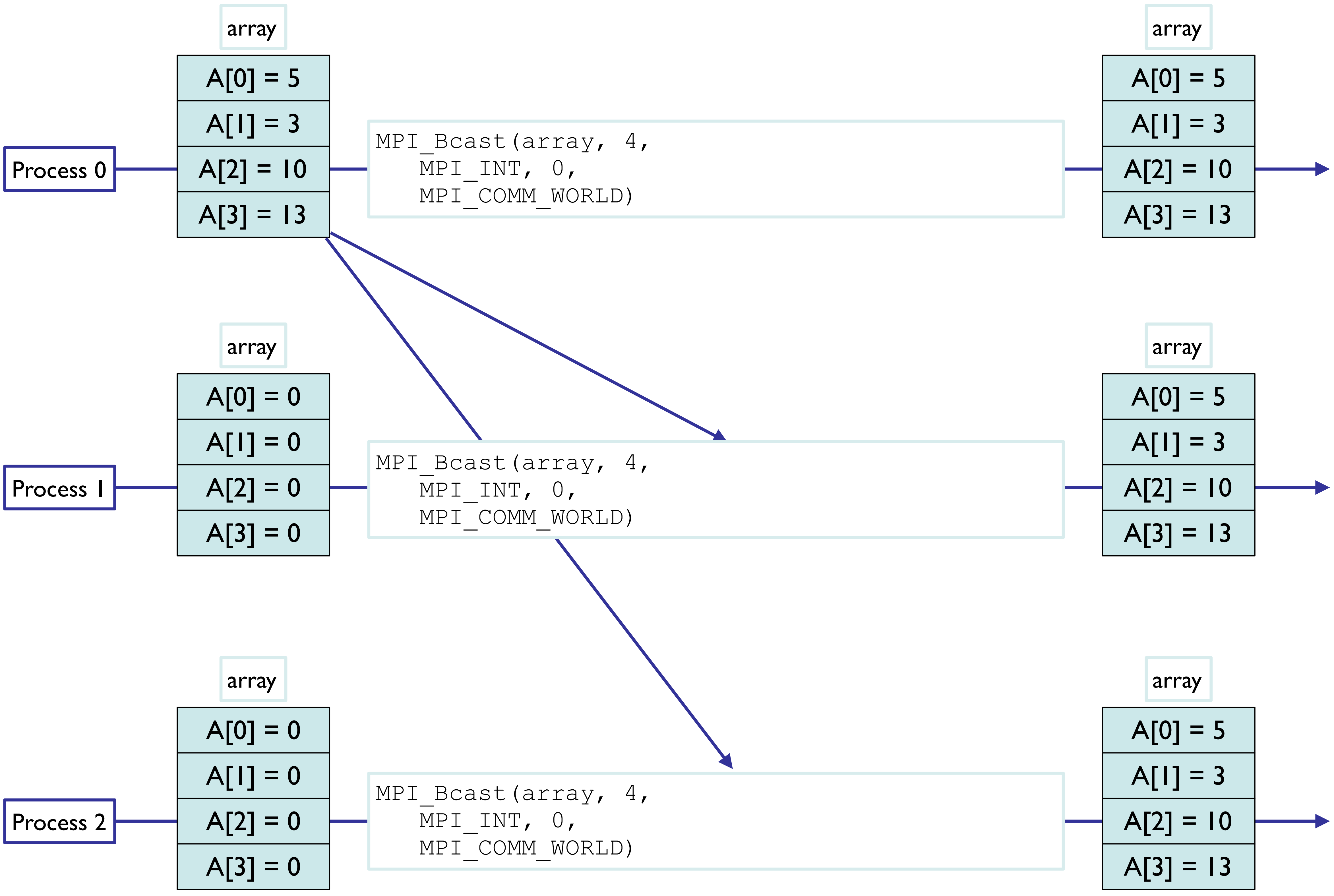
# MPI_Bcast

MPI_Bcast is the simplest of the collective communication methods. It sends a copy of an array to every other process in the communicator passed to it:

```
MPI_Bcast(array /*the data we're broadcasting*/,
          array_size /*the data size */,
          MPI_DOUBLE /*the data type */,
          0 /*the process we're broadcasting from */,
          MPI_COMM_WORLD);
```

Like the rest of the collective communication calls, broadcast is synchronous. The MPI_Bcast function only completes when the process has received all the data.

# MPI_Bcast



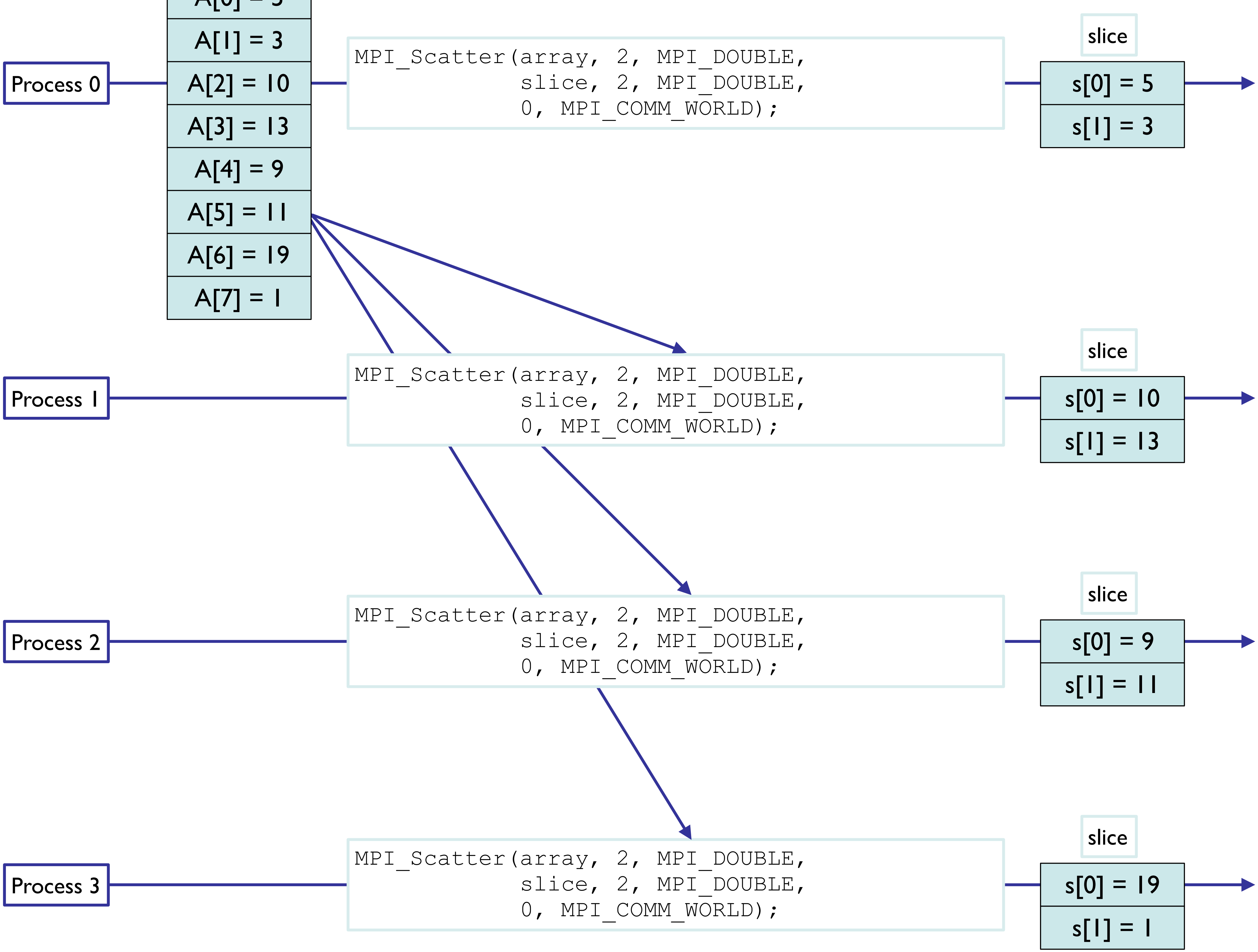
Process 0
Process 1
Process 2
array
A[0] = 5
A[1] = 3
A[2] = 10
A[3] = 13
A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0
MPI_Bcast(array, 4,
MPI_INT, 0,
MPI_COMM_WORLD)

# MPI_Scatter

MPI_Scatter is similar to MPI_Bcast, in that it sends data from one process to every other process. However, in this case we're also splitting up the data, such that each process gets a similarly sized slice. Note that with MPI_Scatter, all the slices are required to be the same size.

```
MPI_Scatter(array       /* the data we're scattering*/,
            slice_size  /* the size of the data we're
                           scattering to each process */,
            MPI_DOUBLE  /* the data type we're sending */,
            array_slice /* where we're receiving the data */,
            slice_size  /* the amount of data we're
                           receiving per process*/,
            MPI_DOUBLE  /* the data type we're receiving */,
            0           /* the process we're sending from*/,
            MPI_COMM_WORLD);
```

# MPI_Scatter

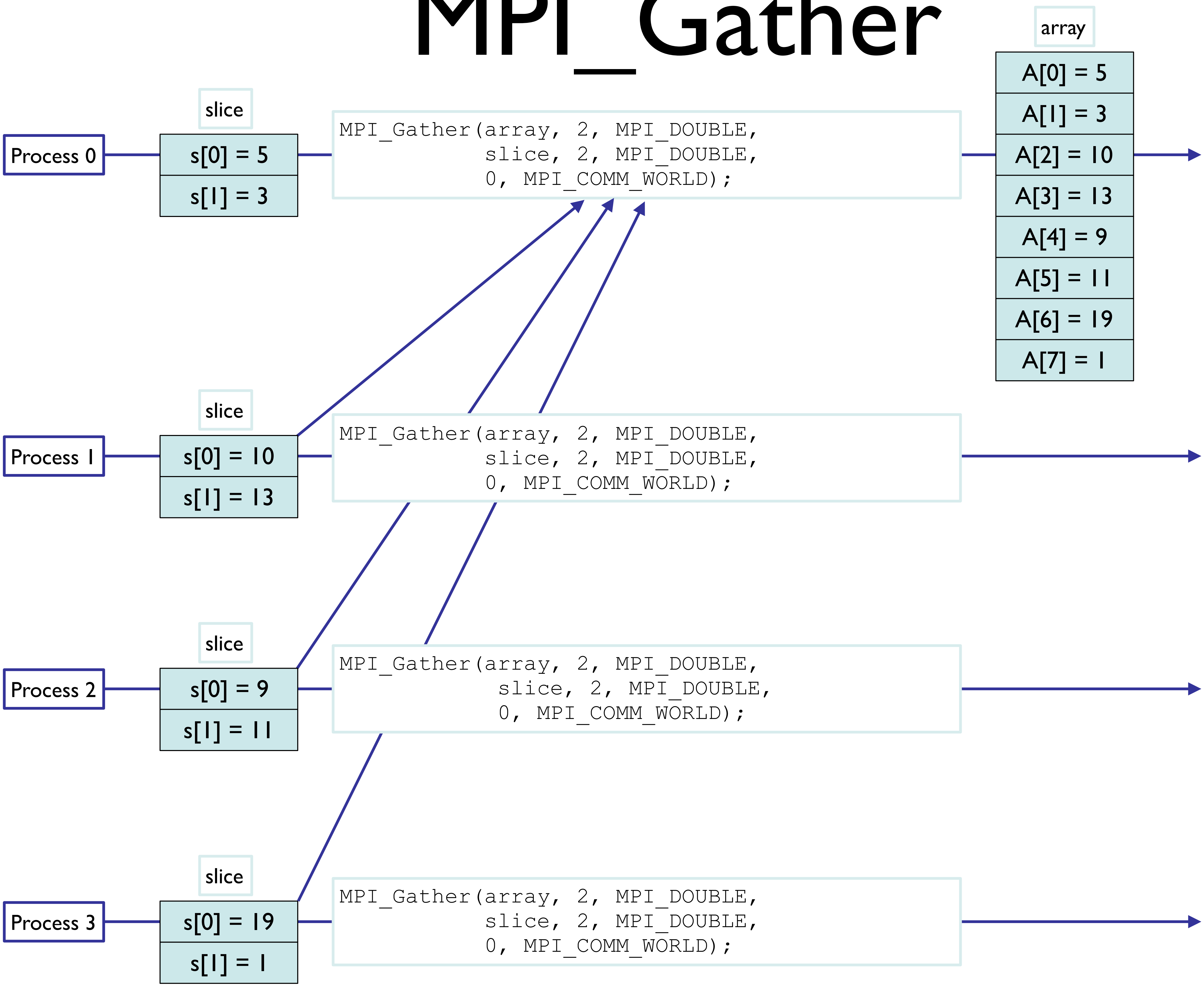

array
A[0] = 5
A[1] = 3
A[2] = 10
A[3] = 13
A[4] = 9
A[5] = 11
A[6] = 19
A[7] = 1
Process 0
MPI_Scatter(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
slice
s[0] = 5
s[1] = 3
Process 1
MPI_Scatter(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
slice
s[0] = 10
s[1] = 13
Process 2
MPI_Scatter(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
slice
s[0] = 9
s[1] = 11
Process 3
MPI_Scatter(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
slice
s[0] = 19
s[1] = 1

# MPI_Gather

MPI_Gather is the opposite of MPI_Scatter. Instead of distributing data from one process to the rest, it takes the slices of data from all the processes and combines it into a single array on the target process.

```
MPI_Gather(array_slice /* where we're receiving the data*/,
           slice_size  /* the size of the data we're
                          receiving */,
           MPI_DOUBLE  /* the data type we're receiving */,
           array       /* the data we're sending */,
           slice_size  /* the amount of data we're
                          sending from each process*/,
           MPI_DOUBLE  /* the data type we're sending */,
           0           /* the process we're receiving to*/,
           MPI_COMM_WORLD);
```

# MPI_Gather


array
A[0] = 5
A[1] = 3
A[2] = 10
A[3] = 13
A[4] = 9
A[5] = 11
A[6] = 19
A[7] = 1
Process 0
slice
s[0] = 5
s[1] = 3
MPI_Gather(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 1
slice
s[0] = 10
s[1] = 13
MPI_Gather(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 2
slice
s[0] = 9
s[1] = 11
MPI_Gather(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 3
slice
s[0] = 19
s[1] = 1
MPI_Gather(array, 2, MPI_DOUBLE,
slice, 2, MPI_DOUBLE,
0, MPI_COMM_WORLD);

# MPI_Scatterv & MPI_Gatherv

MPI_Scatterv and MPI_Gatherv work identically to MPI_Scatter and MPI_Gather, however they allow varying slice sizes.

```
int *array = { 5, 3, 10, 13, 9, 11, 19, 1 };
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };

MPI_Scatterv(array                  /* the data we're scattering*/,
             slice_sizes            /* the size of the data we're
                                       scattering to each process */,
             displacements          /* where the data is going to be sent
                                       from in the array to each process */,
             MPI_DOUBLE             /* the data type we're sending */,
             array_slice            /* where we're receiving the data */,
             slice_sizes[my_rank]   /* the amount of data we’re receiving
                                       per process*/,
             MPI_DOUBLE             /* the data type we're receiving */,
             0                      /* the process we're sending from*/,
             MPI_COMM_WORLD);
```

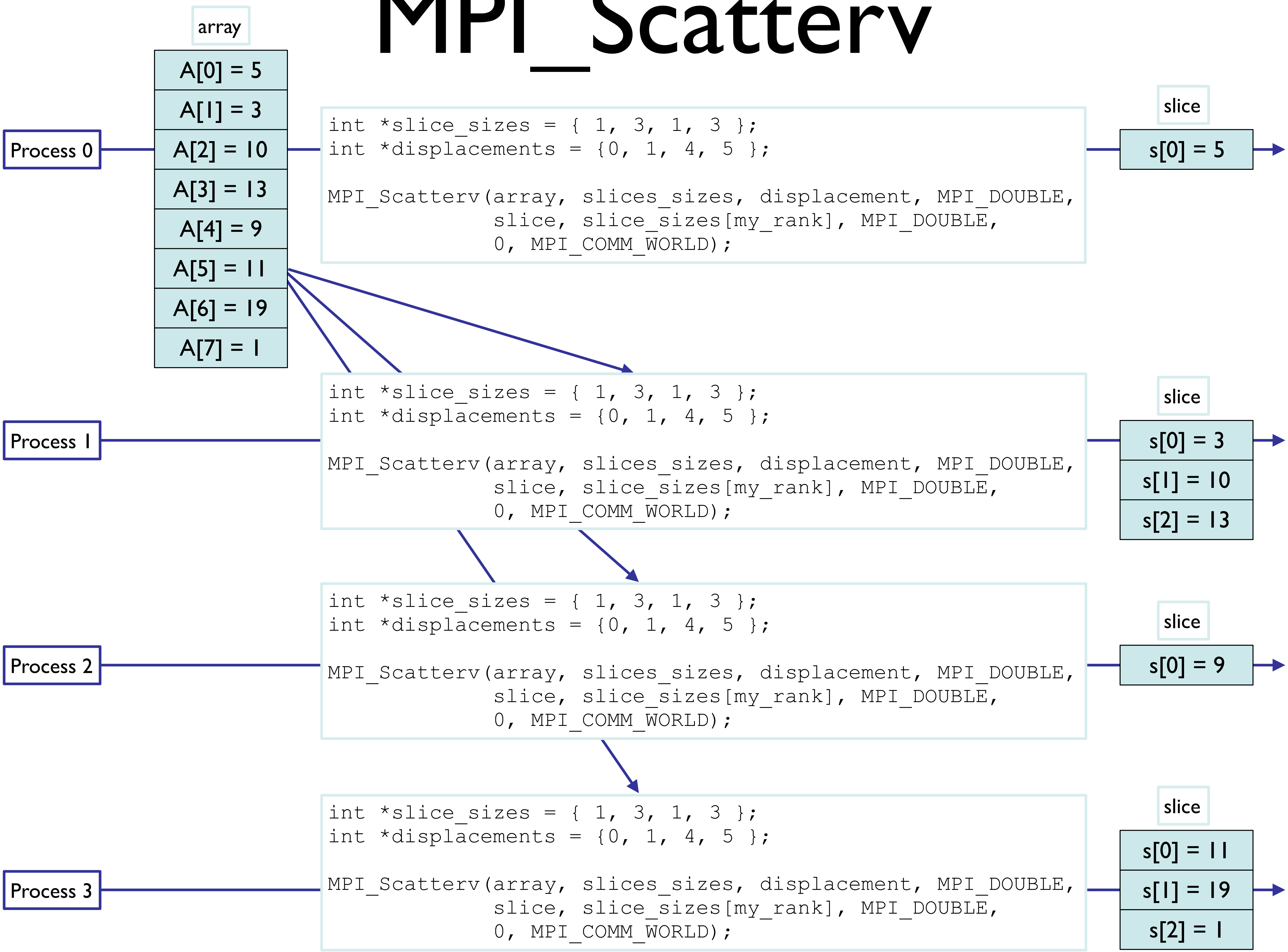
# MPI_Scatterv & MPI_Gatherv

MPI_Scatterv and MPI_Gatherv work identically to MPI_Scatter and MPI_Gather, however they allow varying slice sizes.

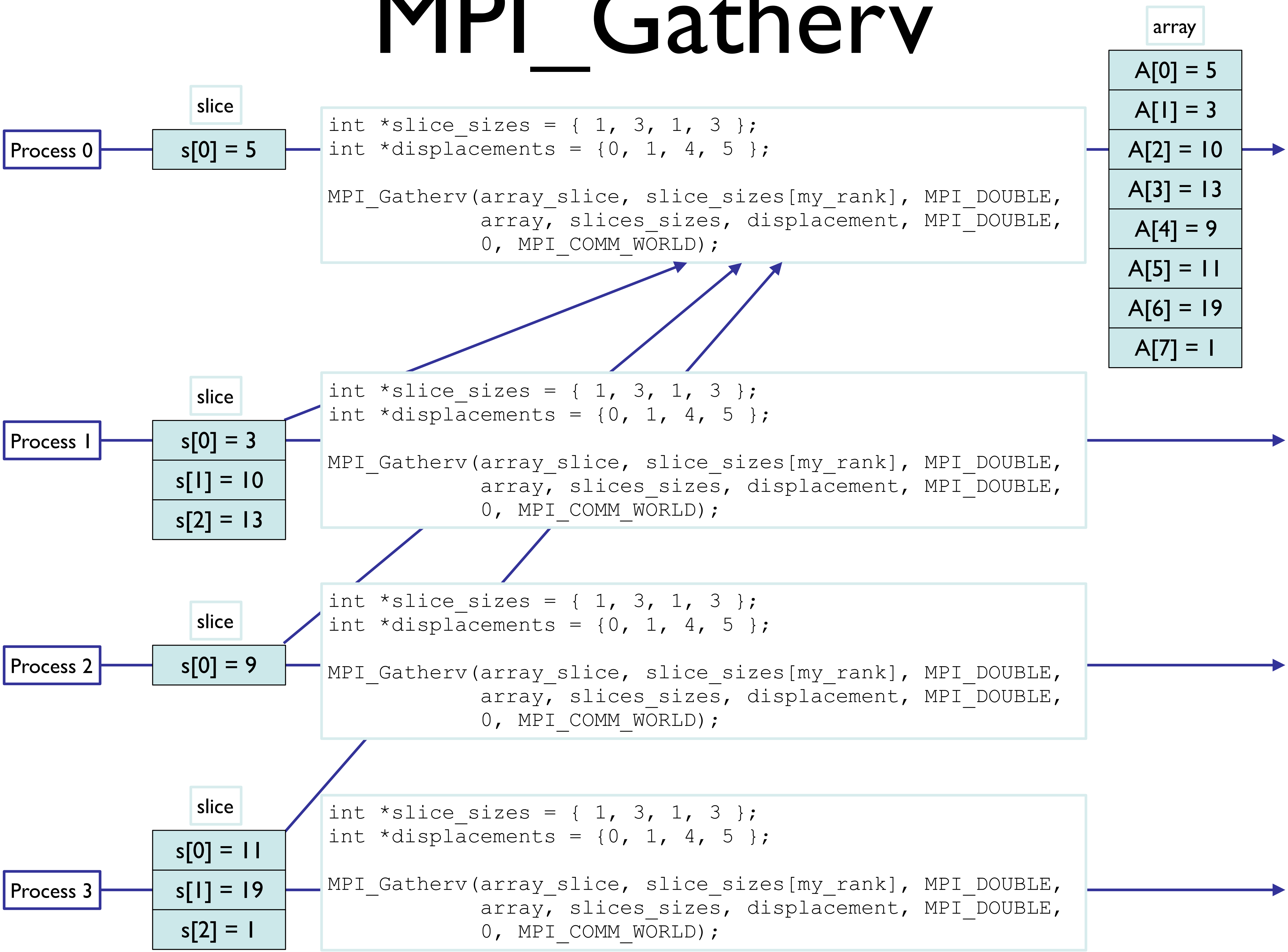
```
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };

MPI_Gatherv( array_slice          /* the data we're gathering*/,
             slice_sizes[my_rank] /* the size of the data we're
                                     sending to the target process */,
             MPI_DOUBLE           /* the data type we're sending */,
             array                /* where we're receiving the data */,
             slice_sizes          /* the amount of data we’re receiving
                                     per process*/,
             displacements        /* where the data from each process is
                                     going to be stored in the array */,
             MPI_DOUBLE           /* the data type we're receiving */,
             0                    /* the process we're sending from*/,
             MPI_COMM_WORLD);
```

# MPI_Scatterv



# MPI_Gatherv


array
A[0] = 5
A[1] = 3
A[2] = 10
A[3] = 13
A[4] = 9
A[5] = 11
A[6] = 19
A[7] = 1
Process 0
slice
s[0] = 5
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };
MPI_Gatherv(array_slice, slice_sizes[my_rank], MPI_DOUBLE,
array, slices_sizes, displacement, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 1
slice
s[0] = 3
s[1] = 10
s[2] = 13
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };
MPI_Gatherv(array_slice, slice_sizes[my_rank], MPI_DOUBLE,
array, slices_sizes, displacement, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 2
slice
s[0] = 9
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };
MPI_Gatherv(array_slice, slice_sizes[my_rank], MPI_DOUBLE,
array, slices_sizes, displacement, MPI_DOUBLE,
0, MPI_COMM_WORLD);
Process 3
slice
s[0] = 11
s[1] = 19
s[2] = 1
int *slice_sizes = { 1, 3, 1, 3 };
int *displacements = {0, 1, 4, 5 };
MPI_Gatherv(array_slice, slice_sizes[my_rank], MPI_DOUBLE,
array, slices_sizes, displacement, MPI_DOUBLE,
0, MPI_COMM_WORLD);

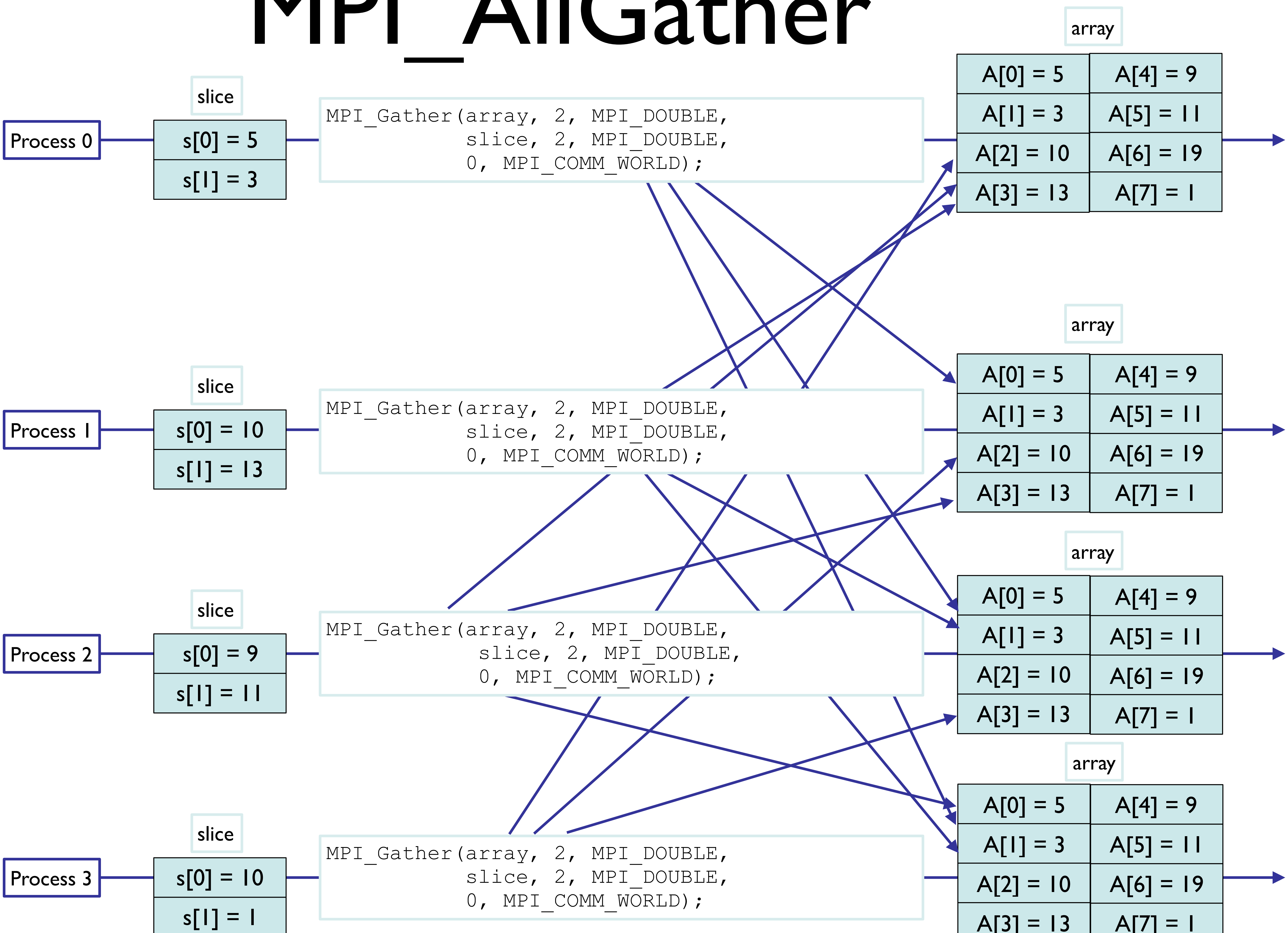
# MPI_Allgather

MPI_Allgather is the same (except more efficiently implemented) as doing MPI_Gather than MPI_Bcast.

There is also an MPI_Allgatherv, which is the same as doing an MPI_Gatherv then MPI_Bcast.

```
MPI_Allgather(array_slice  /* the data we're gathering */,
              slice_size   /* the size of the data we’re
                              gathering */,
              MPI_DOUBLE   /* the data type we're sending */,
              array        /* where we're receiving the data */,
              slice_size   /* the amount of data we're
                              receiving from each process */,
              MPI_DOUBLE   /* the data type we're receiving */,
              MPI_COMM_WORLD);
```

# MPI_AllGather



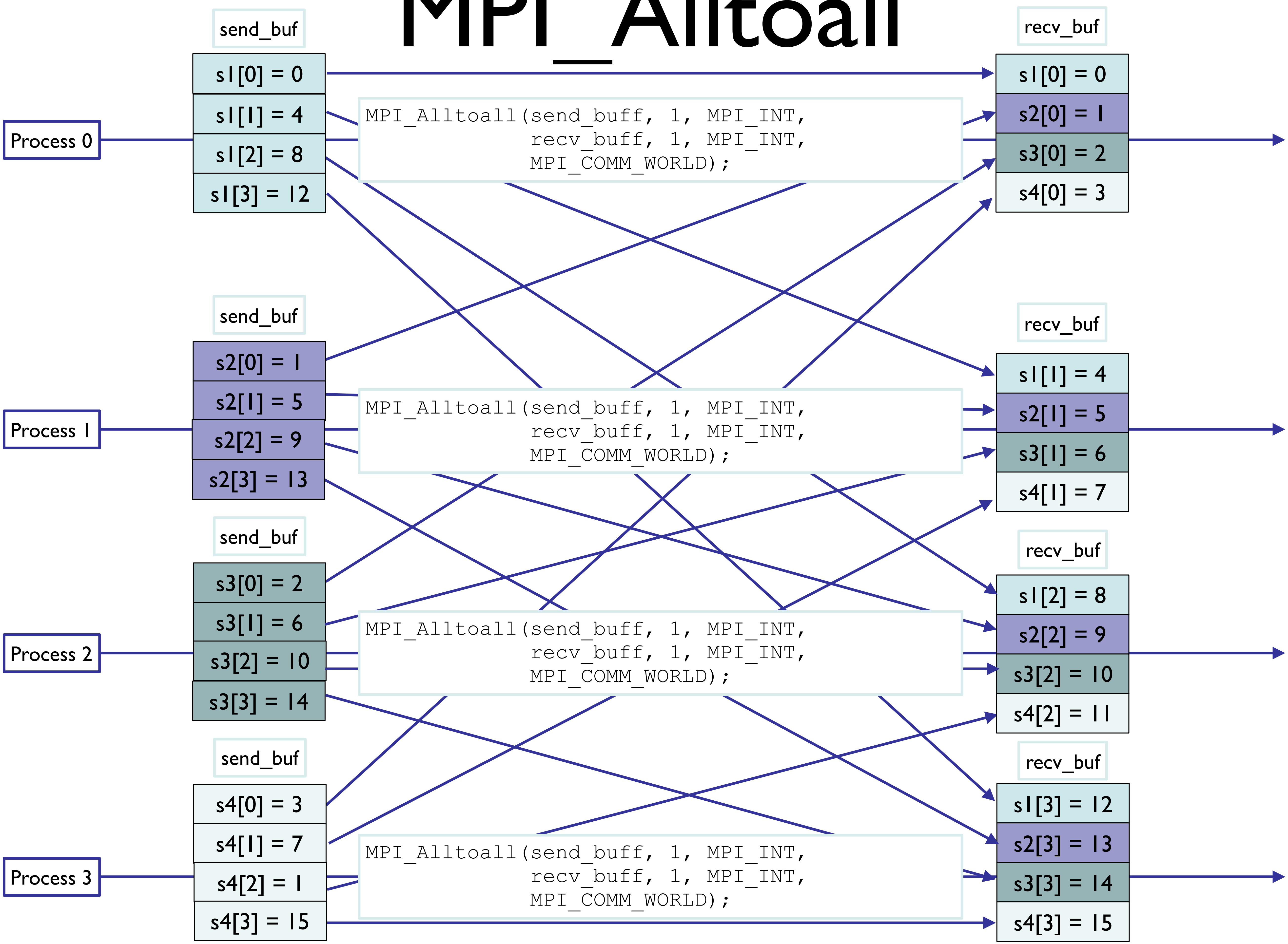
# MPI_Alltoall

All to all (and alltoallv) are another form of collective communication, similar to an all gather. However, in an all to all, each process has a *different* resulting array, getting a different slice from each other process.

```
int MPI_Alltoall(void *sendbuf,   /*the array being sent*/
                 int sendcount,   /*the size of the array being sent*/
                 MPI_Datatype sendtype,
                 void *recvbuf,   /*the array being received into*/
                 int recvcount,   /*the size of the data being received*/
                 MPI_Datatype recvtype,
                 MPI_Comm comm)
```

```
Note that send count should equal recv count, sendbuf and recvbuf should be
different pointers and have memory allocated.
```

# MPI_Alltoall



# Reduce

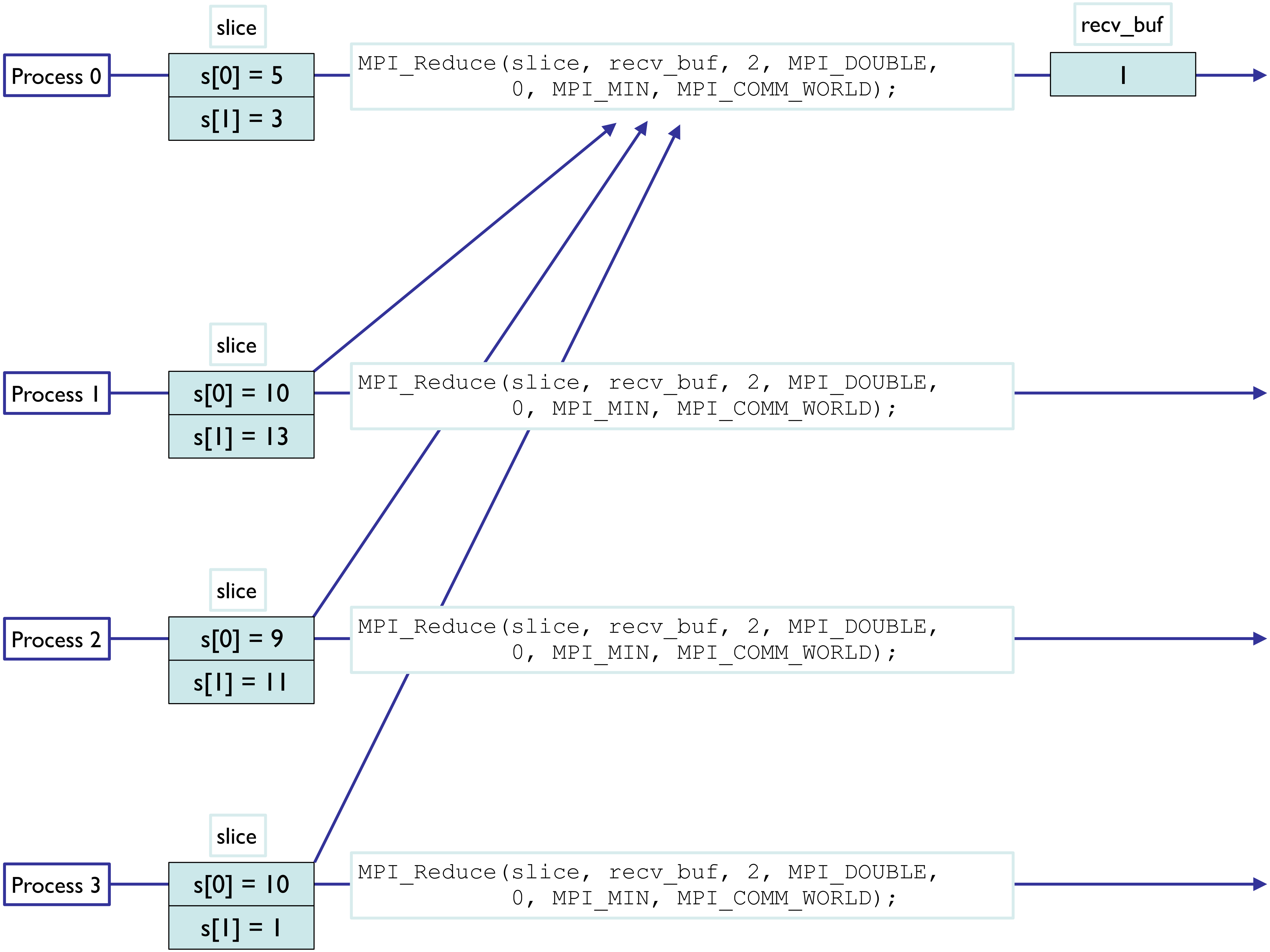
MPI_Reduce applies one of a set of pre-defined operations (like MIN, MAX, SUM, PRODUCT, etc) over the sendbufs across all processes, and returns the result into the recvbuf. The full list of supported operations can be found at:

http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node78.html

There is also an MPI_Allreduce which returns the final value to all processes.

```
int MPI_Reduce(void *sendbuf,
               void *recvbuf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Comm comm)
```

# MPI_Reduce



slice
recv_buf
Process 0
s[0] = 5
s[1] = 3
MPI_Reduce(slice, recv_buf, 2, MPI_DOUBLE, 0, MPI_MIN, MPI_COMM_WORLD);
1
slice
Process 1
s[0] = 10
s[1] = 13
MPI_Reduce(slice, recv_buf, 2, MPI_DOUBLE, 0, MPI_MIN, MPI_COMM_WORLD);
slice
Process 2
s[0] = 9
s[1] = 11
MPI_Reduce(slice, recv_buf, 2, MPI_DOUBLE, 0, MPI_MIN, MPI_COMM_WORLD);
slice
Process 3
s[0] = 10
s[1] = 1
MPI_Reduce(slice, recv_buf, 2, MPI_DOUBLE, 0, MPI_MIN, MPI_COMM_WORLD);

# Asynchronous Communication

# Asynchronous Communication

There are four functions involved in asynchronous communication.

```
MPI_Isend
MPI_Irecv
MPI_Wait
MPI_Test
```

They are most commonly used to be able to overlap communication (which is slow) with computation. This can effectively mask communication overhead and greatly improve performance.

# MPI_Isend

```
MPI_Isend(void* array_pointer,   /*the data we're sending*/
          int array_size,        /*the number of elements*/
          MPI_INT,               /*the data type*/
          int target,            /*the target process*/
          int tag,               /*the message tag*/
          MPI_COMM_WORLD,        /*the communicator*/
          MPI_Request *request); /*an MPI_Request for tracking
                                   when the send is done */
```

# MPI_Irecv

```
MPI_Irecv(void* array_pointer,   /*the data we're receiving*/
          int array_size,        /*the number of elements*/
          MPI_INT,               /*the data type*/
          int source,            /*the source process*/
          int tag,               /*the message tag*/
          MPI_COMM_WORLD,        /*the communicator*/
          MPI_Request *request); /*an MPI_Request for tracking
                                   when the send is done */
```

# MPI_Wait

```
MPI_Request request;
MPI_Irecv(&token, 1, MPI_INT, prev_rank, 0,
          MPI_COMM_WORLD, &request);
…
MPI_Status status;
MPI_Wait(&request, &status);
```

This code is essentially the same as doing an MPI_Recv. MPI_Wait will wait until the message has been completely received before continuing. However with this code we can do calculations in between the MPI_Irecv and the MPI_Wait, where with MPI_Recv, it would block for the communication to complete.

# MPI_Test

```
MPI_Request request;
MPI_Irecv(&token, 1, MPI_INT, prev_rank, 0,
          MPI_COMM_WORLD, &request);

int flag;
MPI_Status status;

while (flag == 0) {
     …
     MPI_Test(&request, &flag, &status);
}
```

This will busy wait until flag != 0, which means the message has been received. MPI_Test exits immediately, so we could potentially do other things in the while loop while we wait for the message to finish being received.

# MPI Datatypes

# Creating MPI Datatypes

```
MPI_Type_create_struct(
  int             count                     /* in */,
  int             array_of_block_lengths[]  /* in */,
  MPI_Aint        array_of_displacements[]  /* in */,
  MPI_Datatype    array_of_types[]          /* in */,
  MPI_Datatype*   new_type_p                /* out */);
```

# Creating MPI Datatypes

```
MPI_Type_create_struct(
  int             count                      /* in */,
  int             array_of_block_lengths[]   /* in */,
  MPI_Aint        array_of_displacements[]   /* in */,
  MPI_Datatype    array_of_types[]           /* in */,
  MPI_Datatype*   new_type_p                 /* out */);

struct {
  double x, y, z, x_vel, y_vel, z_vel;
} Bird;

// none of the elements in our struct are an array
int block_lengths[6] = {1, 1, 1, 1, 1, 1};
// doubles are 8 bytes
int displacements[6] = {0, 8, 16, 24, 32, 40};
// each type is a double
MPI_Datatype types[6] = {MPI_Double, MPI_Double, MPI_Double,
                         MPI_Double, MPI_Double, MPI_Double};
MPI_Datatype *mpi_bird;
MPI_Type_create_struct(6, block_lengths, displacements, types,
                       mpi_bird);
```

# MPI Performance Analysis

```
double MPI_Wtime(void);


double start, finish;

start = MPI_Wtime(); //get a starting time

...

finish = MPI_Wtime(); //get an ending time

printf("[process %d] elapsed time = %e seconds\n",
       my_rank, finish - start);
```