

# Distributed Memory Programming with MPI

Pacheco chapter 3

Moreno Marzolla  
Dip. di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

[moreno.marzolla@unibo.it](mailto:moreno.marzolla@unibo.it)

Copyright © 2013, 2014, 2017–2020  
Moreno Marzolla, Università di Bologna, Italy  
<http://www.moreno.marzolla.name/teaching/HPC/>



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

# Credits

- Peter Pacheco, Dept. of Computer Science, University of San Francisco  
<http://www.cs.usfca.edu/~peter/>
- Mary Hall, School of Computing, University of Utah  
<https://www.cs.utah.edu/~mhall/>
- Salvatore Orlando, Univ. Ca' Foscari di Venezia  
<http://www.dsi.unive.it/~orlando/>
- Blaise Barney,  
<https://computing.llnl.gov/tutorials/mpi/> (**highly recommended!!**)

# Introduction

# Message Passing and MPI

- **Message passing** is the principal alternative to shared memory parallel programming
  - Message passing represents the predominant programming model for supercomputers and clusters
- What MPI is
  - A library used within conventional sequential languages (Fortran, C, C++)
  - Based on Single Program, Multiple Data (SPMD)
  - Isolation of separate address spaces
    - no data races, but communication errors possible
    - exposes execution model and forces programmer to think about locality, both good for performance
    - complexity and code growth!

# SPMD

## (Single Program Multiple Data)

- The **same** program is executed by  $P$  processes
- Each process may choose a different execution path depending on its ID (*rank*)

```
...
MPI_Init(...);
...
foo();           /* executed by all processes */
if ( my_id == 0 ) {
    do_something(); /* executed by process 0 only */
} else {
    do_something_else(); /* executed by all other processes */
}
...
MPI_Finalize();
...
```

# Message Passing and MPI

- All communication and synchronization operations require subroutine calls
  - No shared variables
- Subroutines for
  - **Communication**
    - Pairwise or point-to-point
    - Collectives involving multiple processes
  - **Synchronization**
    - Barrier
    - No locks because there are no shared variables to protect
  - **Queries**
    - How many processes? Which one am I? Any messages waiting?

# Using MPI under Debian/Ubuntu

- Install the `mpi-default-bin` and `mpi-default-dev` packages
  - Installs OpenMPI
  - You might also want `openmpi-doc` for the man pages
- Use `mpicc` to compile, `mpirun` to execute
- To execute your program on remote hosts, make sure you can ssh into them without entering a password
  - Generate a public/private key pair on your local machine, if you have not already done so, with `ssh-keygen -t dsa`; do not enter a passphrase
  - Append the content of `.ssh/id_dsa.pub` to the remote file `.ssh/authorized_keys`



# Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Who am I?
- MPI provides functions to answer these questions:
  - `MPI_Comm_size` reports the number of processes
  - `MPI_Comm_rank` reports the rank, a number between 0 and  $(size - 1)$ , identifying the calling process

# Hello, world!

```
/* mpi-hello.c */

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

int main( int argc, char *argv[] )
{
    int rank, size, len;
    char hostname[MPI_MAX_PROCESSOR_NAME];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Get_processor_name( hostname, &len );
    printf("Greetings from process %d of %d running on %s\n",
           rank, size, hostname);
    MPI_Finalize();
    return 0;
}
```

*No MPI call before this line*

*No MPI call after this line*

# Hello, world!

- Compilation:

```
mpicc -Wall mpi_hello.c -o mpi_hello
```

- Execution (8 processes on localhost):

```
mpirun -n 8 ./mpi_hello
```

- Execution (two processes on host “foo” and one on host “bar”)

```
mpirun -H foo,foo,bar ./mpi_hello
```

# Hello, world!

```
$ mpirun -n 8 ./mpi_hello
```

```
Greetings from process 7 of 8 running on wopr
```

```
Greetings from process 5 of 8 running on wopr
```

```
Greetings from process 0 of 8 running on wopr
```

```
Greetings from process 3 of 8 running on wopr
```

```
Greetings from process 6 of 8 running on wopr
```

```
Greetings from process 4 of 8 running on wopr
```

```
Greetings from process 1 of 8 running on wopr
```

```
Greetings from process 2 of 8 running on wopr
```

# Hostfile

```
$ cat myhostfile  
aa slots=4  
bb slots=4  
cc slots=4
```

- To run 4 instances on node “aa” and 2 instances on node “bb”:

```
$ mpirun -hostfile myhostfile -n 6 ./mpi_hello
```

- To run 2 instances on “aa”, 2 on “bb” and the remaining 2 on “cc”:

```
$ mpirun -loadbalance -hostfile myhostfile -n 6 ./mpi_hello
```

- **man mpirun** for additional information

# Oversubscribing

- Recent versions of OpenMPI complain if you try to launch more processes than the number of processor core(s) available
- To overcome this you need to pass the **--oversubscribe** command line option to mpirun

```
$ mpirun --oversubscribe -n 30 ./mpi_hello
```

# MPI

- The following six functions suffice for most programs:
  - `MPI_Init`
  - `MPI_Finalize`
  - `MPI_Comm_size`
  - `MPI_Comm_rank`
  - `MPI_Send` (blocking send)
  - `MPI_Recv` (blocking receive)
  - `MPI_Abort` (aborting the computation)

# A Simple MPI Program

```
/* mpi-point-to-point.c */
#include <mpi.h>
#include <stdio.h>
int main( int argc, char *argv[])
{
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* process 0 sends and process 1 receives */
    if (rank == 0) {
        buf = 123456;
        MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        printf("Received %d\n", buf);
    }

    MPI_Finalize();
    return 0;
}
```



# Some Basic Concepts

- How to organize processes
  - Processes can be collected into **groups**
  - A group and context together form a **communicator**
  - A process is identified by its rank in the group associated with a communicator
- There is a default communicator **MPI\_COMM\_WORLD** whose group contains all processes

# MPI datatypes

- The data to be sent or received is described by a triple (**address**, **count**, **datatype**), where an MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., **MPI\_INT**, **MPI\_DOUBLE**)
  - a contiguous array of MPI datatypes
  - a *strided* block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes

# Some MPI datatypes

<b>MPI Datatype</b>	<b>C datatype</b>
<code>MPI_CHAR</code>	signed char
<code>MPI_SHORT</code>	signed short int
<code>MPI_INT</code>	signed int
<code>MPI_LONG</code>	signed long int
<code>MPI_LONG_LONG</code>	signed long long int
<code>MPI_UNSIGNED_CHAR</code>	unsigned char
<code>MPI_UNSIGNED_SHORT</code>	unsigned short int
<code>MPI_UNSIGNED</code>	unsigned int
<code>MPI_UNSIGNED_LONG</code>	unsigned long int
<code>MPI_FLOAT</code>	float
<code>MPI_DOUBLE</code>	double
<code>MPI_LONG_DOUBLE</code>	long double
<code>MPI_BYTE</code>	

# MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **MPI\_ANY\_TAG** as the tag in a receive

# MPI Basic (Blocking) Send

```
int buf = 123456;
MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
int MPI_Send(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

- The message buffer is described by (buf, count, datatype).
- count is the number of *items* to send (NOT the number of bytes)
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm
  - If dest is MPI\_PROC\_NULL, the MPI\_Send operation has no effect
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may have not been received yet by the target process.

# MPI Basic (Blocking) Receive

```
MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,  
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is the process rank in the communicator specified by comm, or **MPI\_ANY\_SOURCE**
- **tag** is a tag to be matched, or **MPI\_ANY\_TAG**
- receiving fewer than **count** occurrences is OK, but receiving more is an error
- **status** contains further information (e.g. size of message), or **MPI\_STATUS\_IGNORE** if no information is needed

# MPI\_Status

- **MPI\_Status** is a C structure with (among others) the following fields:
  - `int MPI_SOURCE;`
  - `int MPI_TAG;`
  - `int MPI_ERROR;`
- Therefore, a process can check the actual source and tag of a message received with **MPI\_ANY\_TAG** or **MPI\_ANY\_SOURCE**

# MPI\_Get\_count()

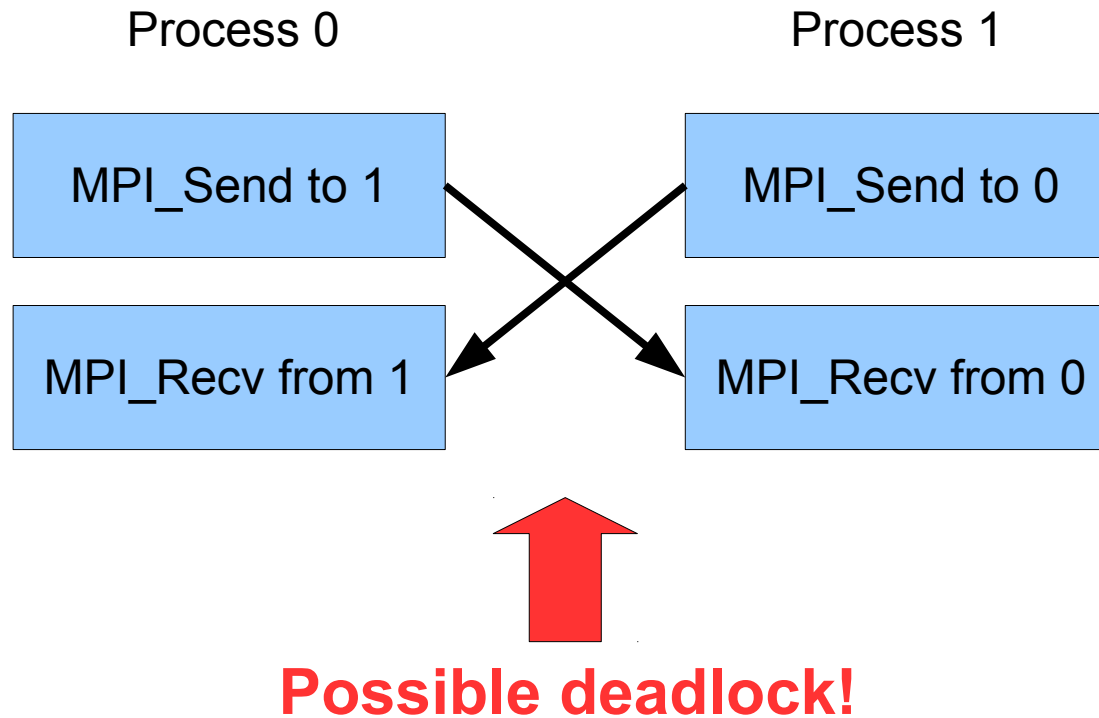
```
int MPI_Get_count( const MPI_Status *status,  
MPI_Datatype datatype, int *count )
```

- **MPI\_Recv** may complete even if less than *count* elements have been received
  - Provided that the matching **MPI\_Send** actually sent fewer elements
- **MPI\_Get\_count** can be used to know how many elements of type *datatype* have actually been received
- See [mpi-get-count.c](#)

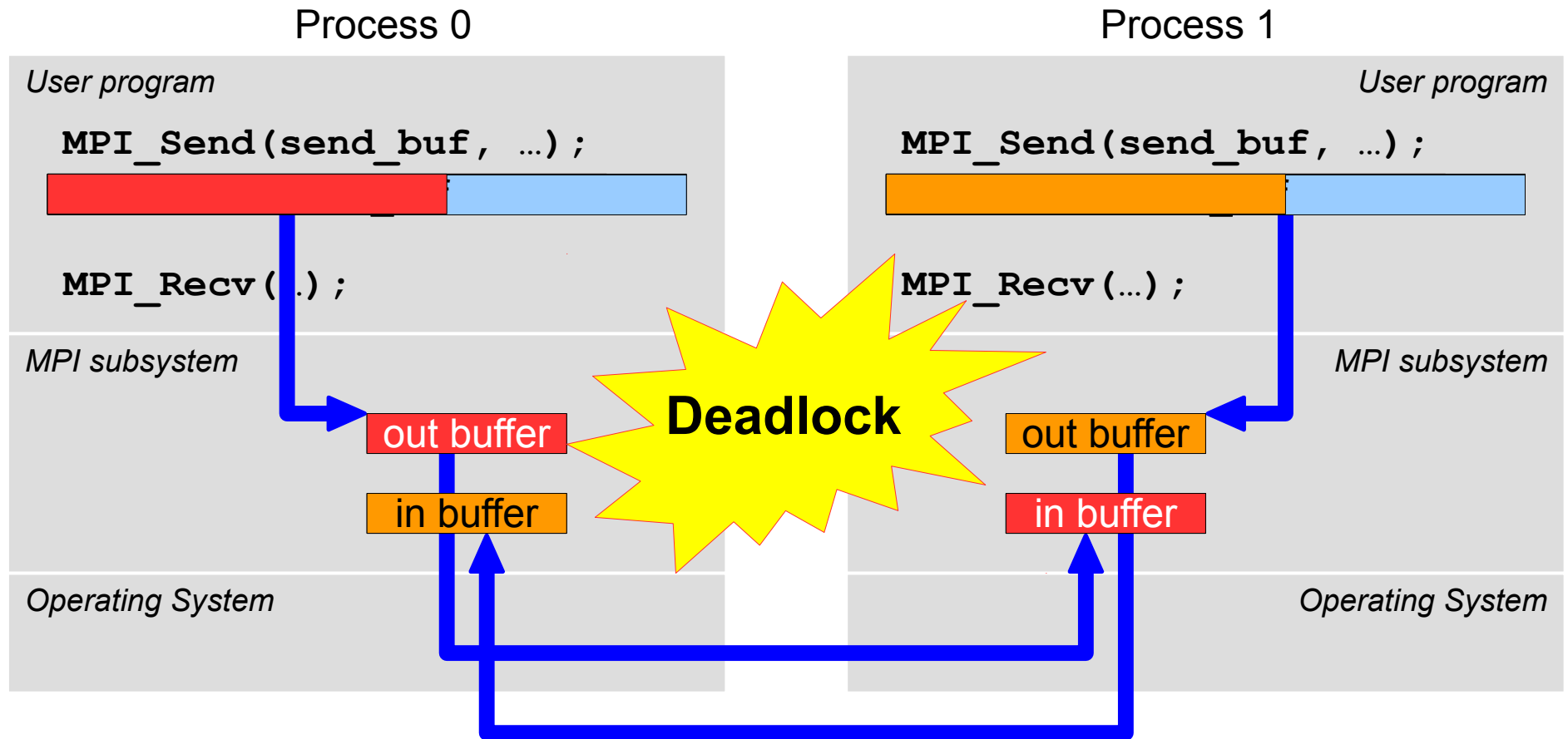


# Blocking communication and deadlocks

- Blocking send/receive may lead to deadlock if not paired carefully

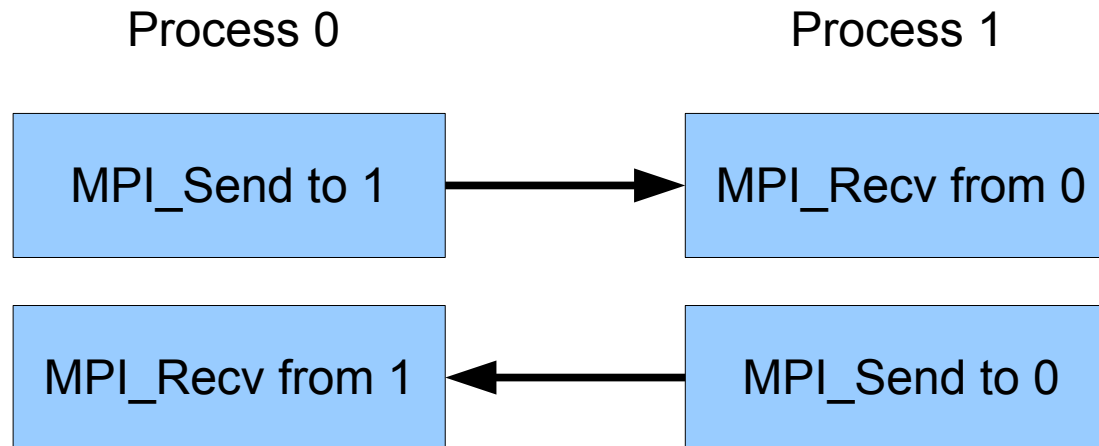


# Blocking communication and deadlocks



# Blocking communication and deadlocks

- To avoid the deadlock it is necessary to reorder the operations so that send/receive pairs match...



- ...or use **non-blocking** communication primitives

# Non-blocking Send

```
int buf = 123456;
MPI_Request req;
MPI_Isend(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
```

```
int MPI_Isend(const void *start, int count,
MPI_Datatype datatype, int dest, int tag, MPI_Comm
comm, MPI_Request *req)
```

- The message buffer is described by (start, count, datatype).
- count is the number of *items* to send
- The target process is specified by dest, which is the rank of the target process in the communicator specified by comm
- A unique identifier of this request is stored to req
- This function returns immediately

# Non-blocking Receive

```
MPI_Request req;  
MPI_Irecv(&buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req);
```

```
int MPI_Irecv(void *start, int count, MPI_Datatype  
datatype, int source, int tag, MPI_Comm comm,  
MPI_Request *req)
```

- Processing continues immediately without waiting for the message to be received
- A communication request handle (`req`) is returned for handling the pending message status
- The program must call `MPI_Wait()` or `MPI_Test()` to determine when the non-blocking receive operation completes
- **Note:** it is OK to use `MPI_Isend` with the (blocking) `MPI_Recv`, and vice-versa

# MPI\_Test()

```
int MPI_Test(MPI_Request *request, int *flag,  
MPI_Status *status)
```

- Checks the status of a specified non-blocking send or receive operation
- The integer `flag` parameter is set to 1 if the operation has completed, 0 if not
- For multiple non-blocking operations, there exist functions to specify any (`MPI_Testany`), all (`MPI_Testall`) or some (`MPI_Testsome`) completions
- See man pages for details

# MPI\_Wait()

```
int MPI_Wait(MPI_Request *request, MPI_Status
*status)
```

- Blocks until a specified non-blocking send or receive operation has completed
- For multiple non-blocking operations, there exists variants to specify any (`MPI_Waitany`), all (`MPI_Waitall`) or some (`MPI_Waitsome`) completions
- See man pages for details

# Async send demo

```
/* mpi-async.c */
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[])
{
    int rank, size, buf;
    MPI_Status status;
    MPI_Request req;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    if (rank == 0) {
        buf = 123456;
        MPI_Isend( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
        big_computation();
        MPI_Wait(&req, &status);
    } else if (rank == 1) {
        MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
        printf("Received %d\n", buf);
    }

    MPI_Finalize();
    return 0;
}
```

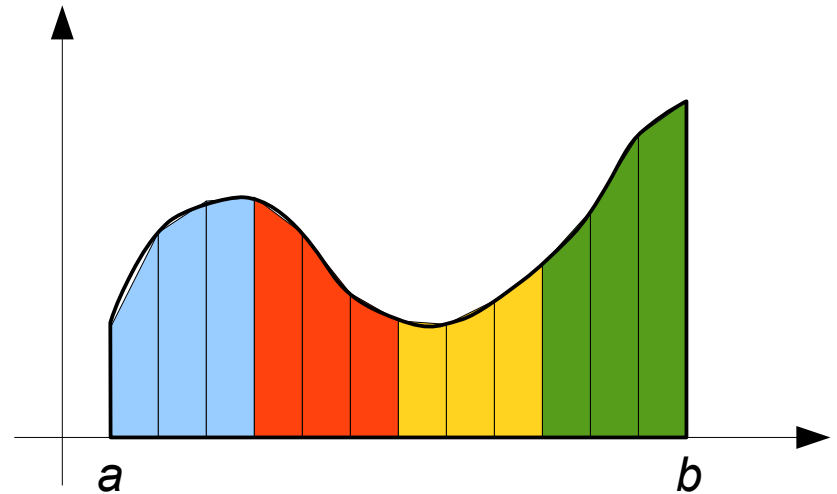
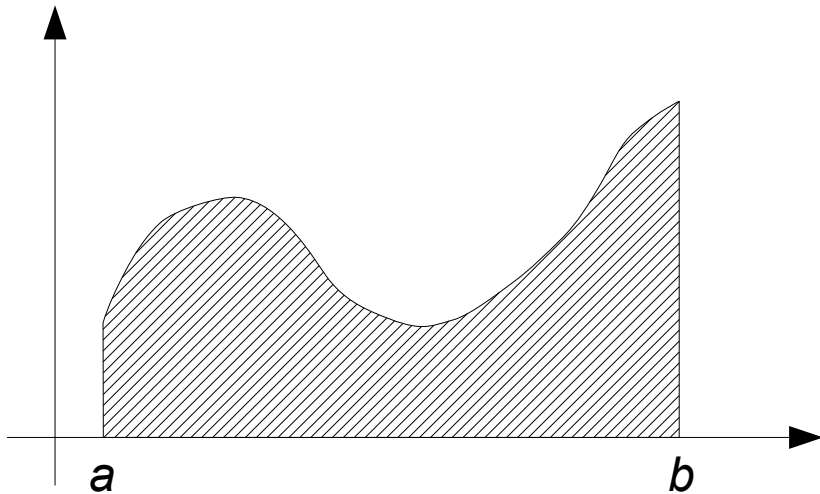


# Aborting the computation

- To abort a computation, do not use `exit()` or `abort()`: call `MPI_Abort()` instead
- `MPI_Abort(comm, err)` "gracefully" terminates all running MPI processes on communicator "comm" (e.g., `MPI_COMM_WORLD`) returning the error code "err"

# Example: Trapezoid rule with MPI

# The trapezoid strikes back



```
double result = 0.0;
double h = (b-a)/n;
double x = a;
int i;
for ( i = 0; i<n; i++ ) {
    result += h*(f(x) + f(x+h))/2.0;
    x += h;
}
```

See [trap.c](#)

# Parallel pseudo-code (naïve)

```
partial_result = trap(my_rank, comm_sz, a, b, n);

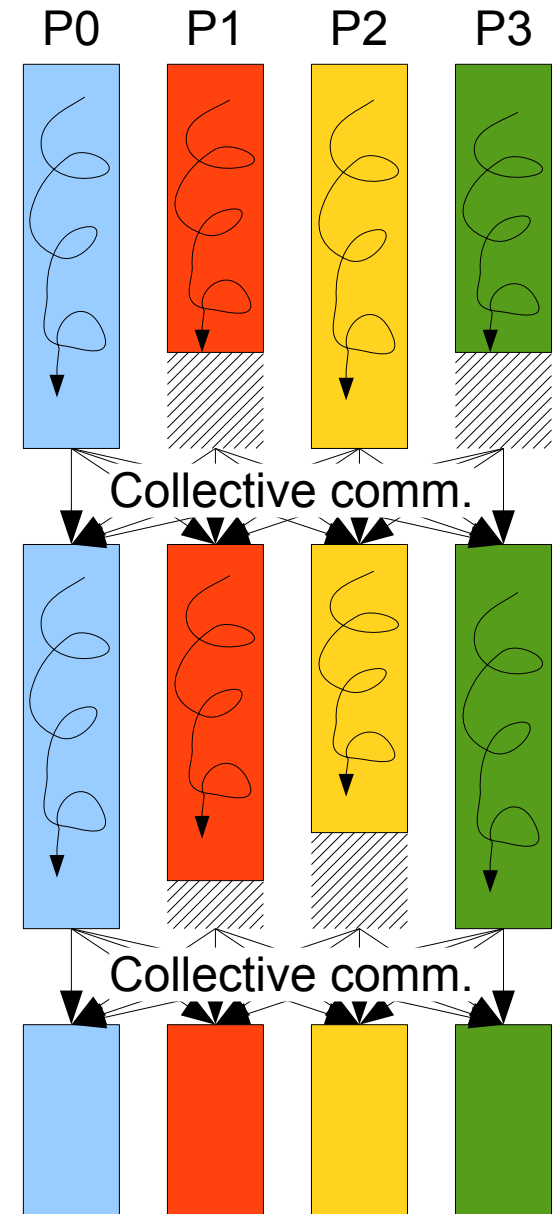
if (my_rank != 0) {
    Send partial_result to process 0;
} else { /* my_rank == 0 */
    result = partial_result;
    for ( p = 1; p < comm_sz; p++ ) {
        Receive partial_result from process p;
        result += partial_result;
    }
    print result;
}
```

See [mpi-trap0.c](#)

# Collective Communication

# Collective communications

- Send/receive operations are rarely used in practice
- Many applications use the *bulk synchronous* pattern:
  - Repeat:
    - Local computation
    - Communicate to update global view on all processes
- Collective communications are executed by all processes in the group to compute and share some global result



# Collective communications

- Collective communications are assumed to be more efficient than point-to-point operations achieving the same result
- Understanding when collective communications are to be used is an **essential skill** of a MPI programmer

# MPI\_Barrier()

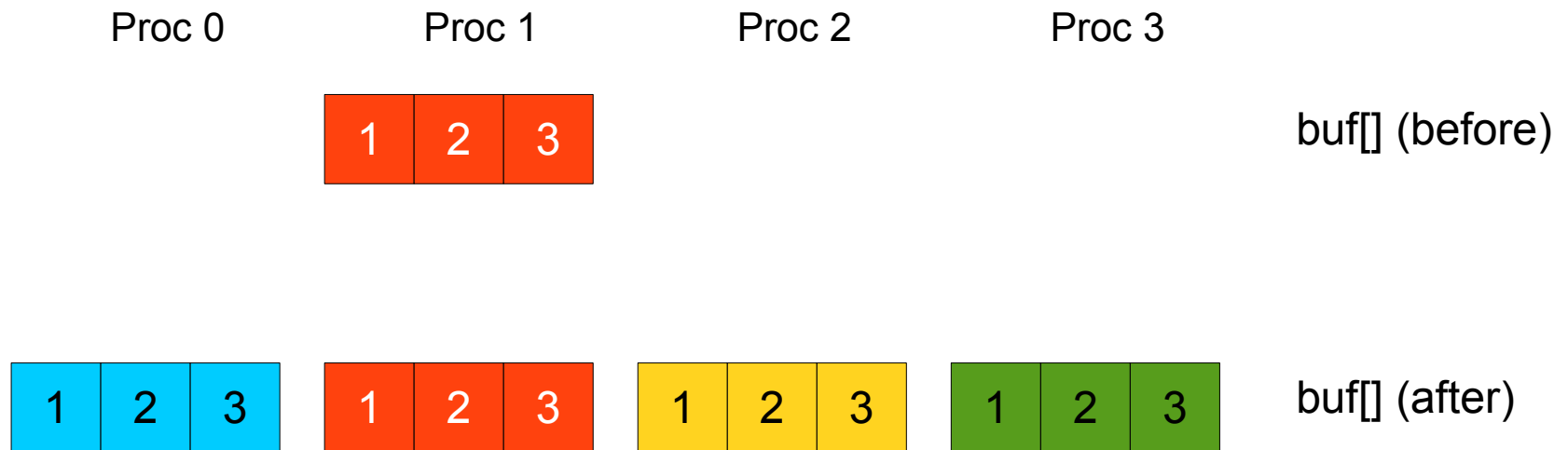
- Executes a barrier synchronization in a group
  - When reaching the `MPI_Barrier()` call, a process blocks until all processes in the group reach the same `MPI_Barrier()` call
  - Then all processes are free to continue



# MPI\_Bcast()

Broadcasts a message to all other processes of a group

```
count = 3;  
src = 1; /* broadcast originates from process 1 */  
MPI_Bcast(buf, count, MPI_INT, src, MPI_COMM_WORLD);
```



# MPI\_Scatter()

Distribute data to other processes in a group

```
sendcnt = 3; /* how many items are sent to each process */
recvcnt = 3; /* how many items are received by each process */
src = 1; /* process 1 contains the message to be scattered */
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT, src, MPI_COMM_WORLD);
```

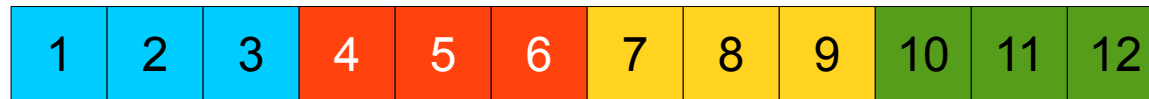
sendbuf[] (before)

recvbuf[] (after)

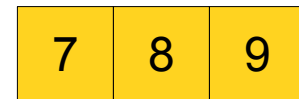
Proc 0



Proc 1



Proc 2



Proc 3



# MPI\_Scatter()

- The **MPI\_Scatter()** operation produces the same result as if the root executes a series of

```
MPI_Send(sendbuf + i * sendcount * extent(sendtype),  
sendcount, sendtype, i, ...)
```

and all other processes execute

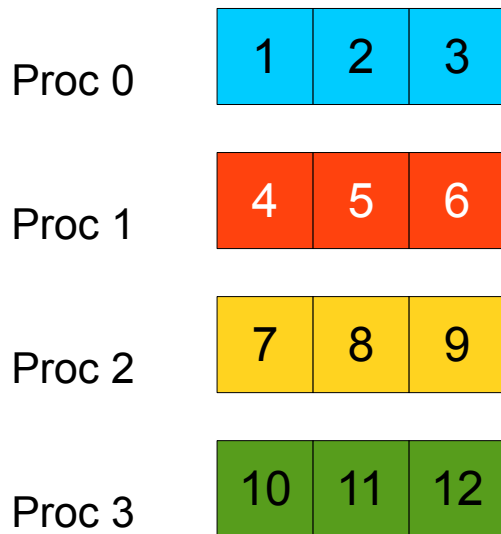
```
MPI_Recv(recvbuf, recvcount, recvtype, i, ...)
```

# MPI\_Gather()

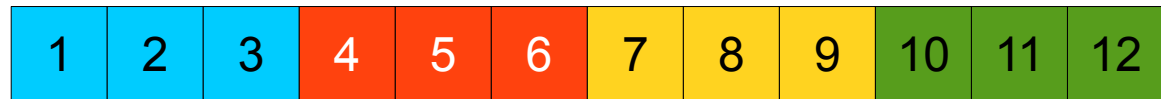
Gathers together data from other processes

```
sendcnt = 3; /* how many items are sent by each process */
recvcnt = 3; /* how many items are received from each process */
dst = 1;    /* message will be gathered at process 1 */
MPI_Gather(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT, dst, MPI_COMM_WORLD);
```

sendbuf[] (before)



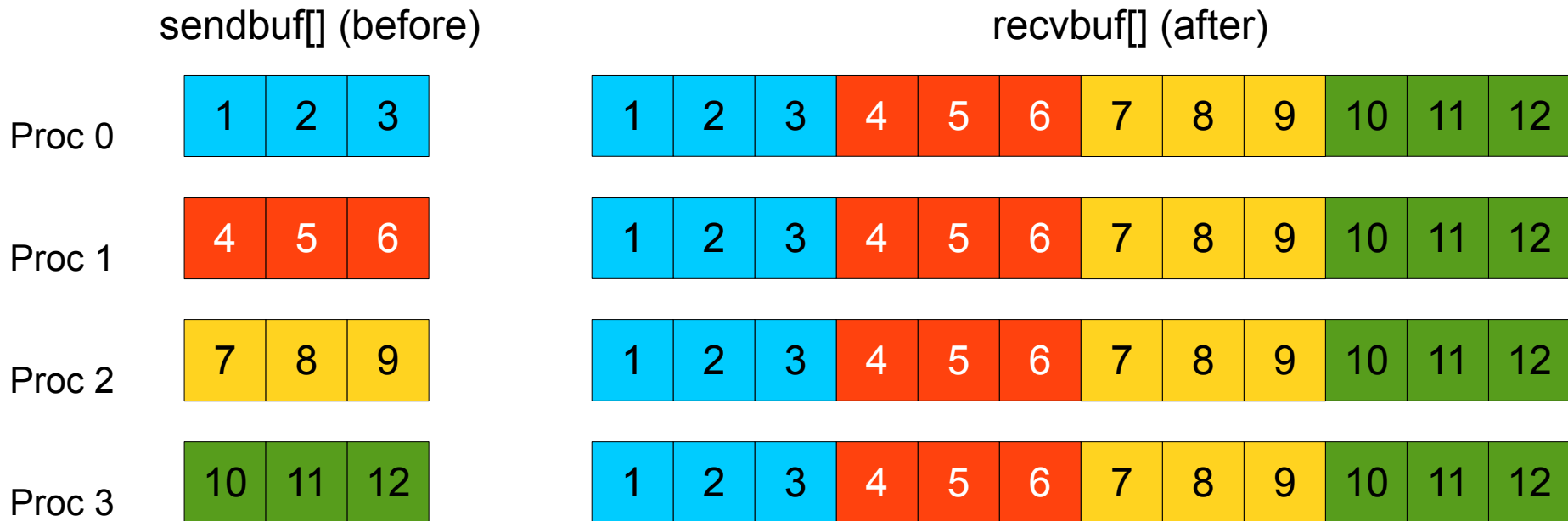
recvbuf[] (after)



# MPI\_Allgather()

Gathers data from other processes and distribute to all

```
sendcnt = 3;  
recvcnt = 3;  
MPI_Allgather(sendbuf, sendcnt, MPI_INT,  
               recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
```

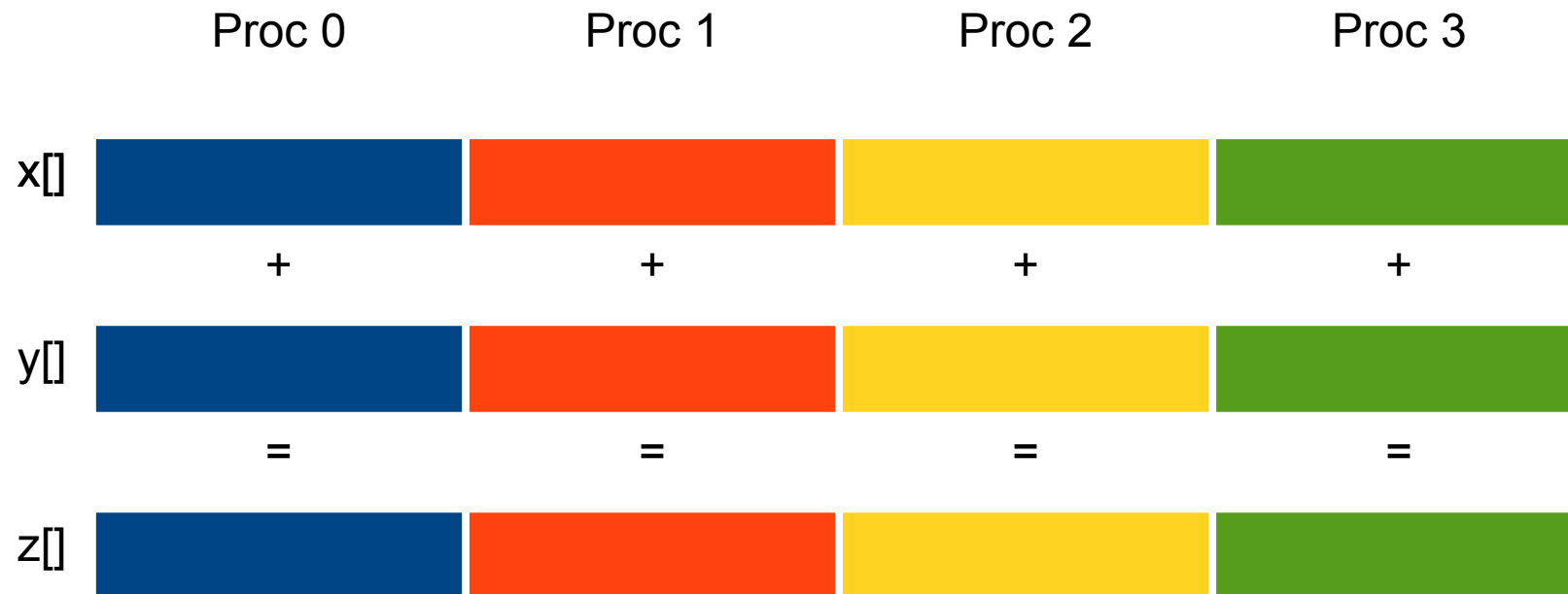


# Example: Parallel Vector Sum

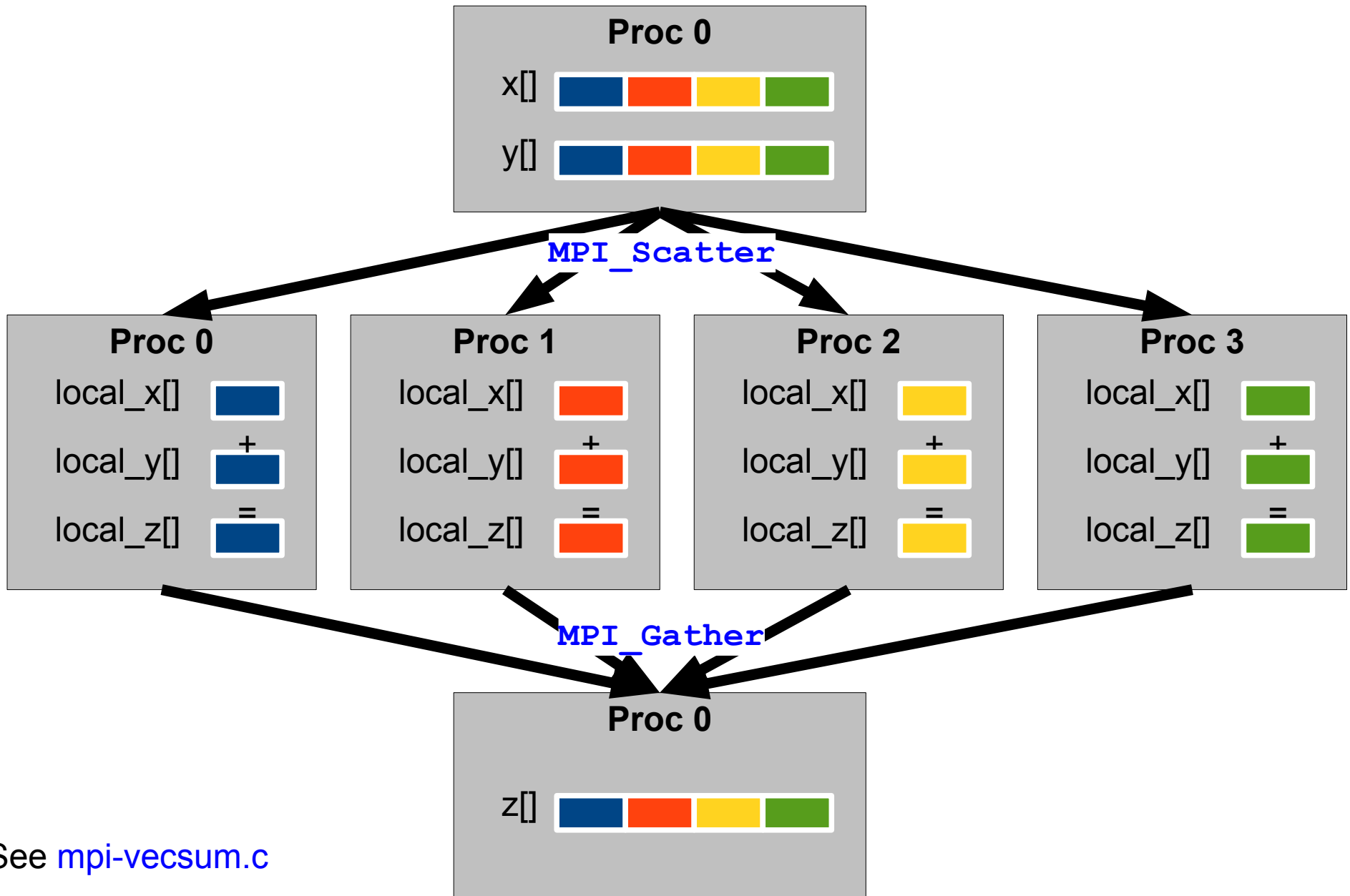
$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

```
void sum( double* x, double* y, double* z, int n )
{
    int i;
    for (i=0; i<n; i++) {
        z[i] = x[i] + y[i];
    }
}
```

# Parallel Vector Sum



# Parallel Vector Sum

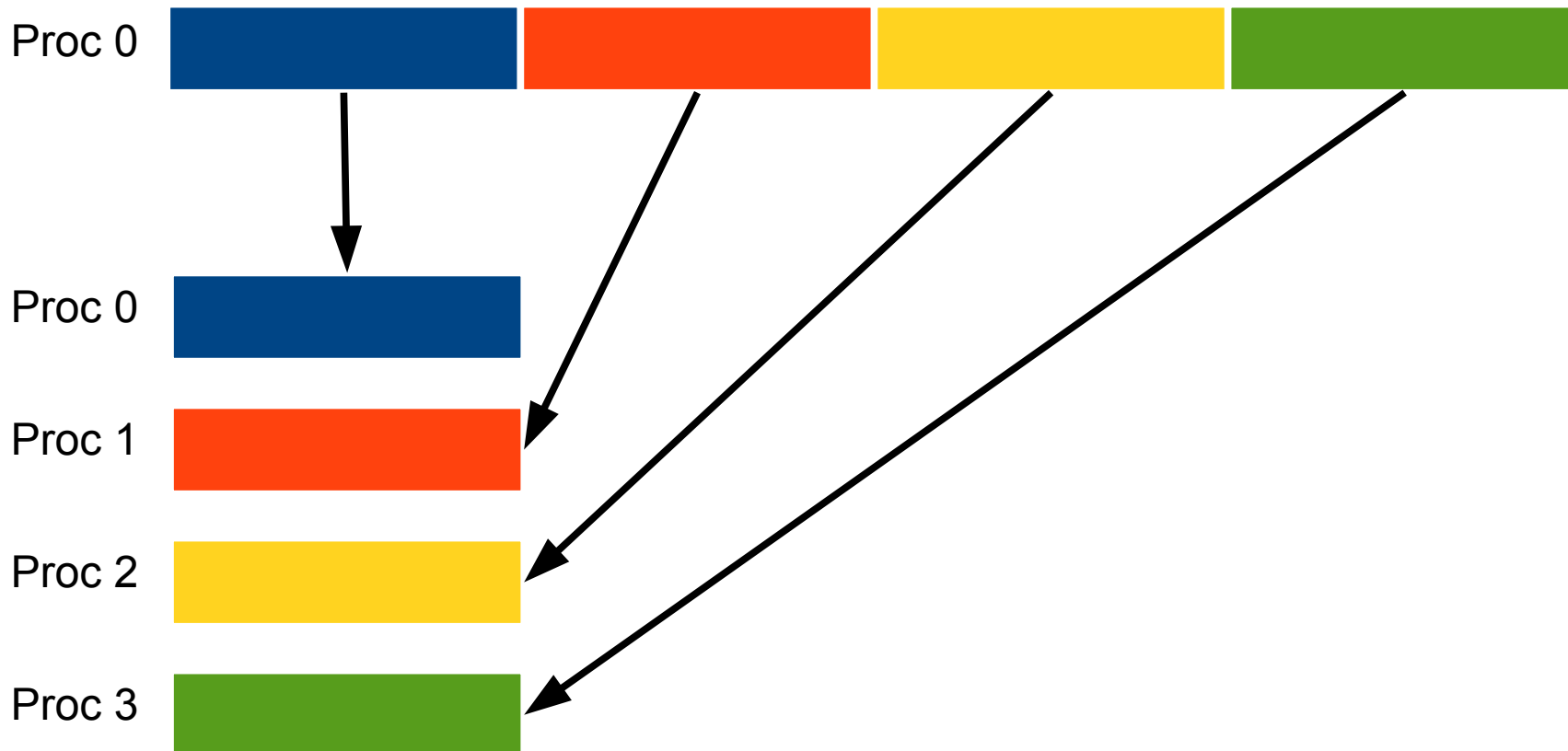


See [mpi-vecsum.c](#)



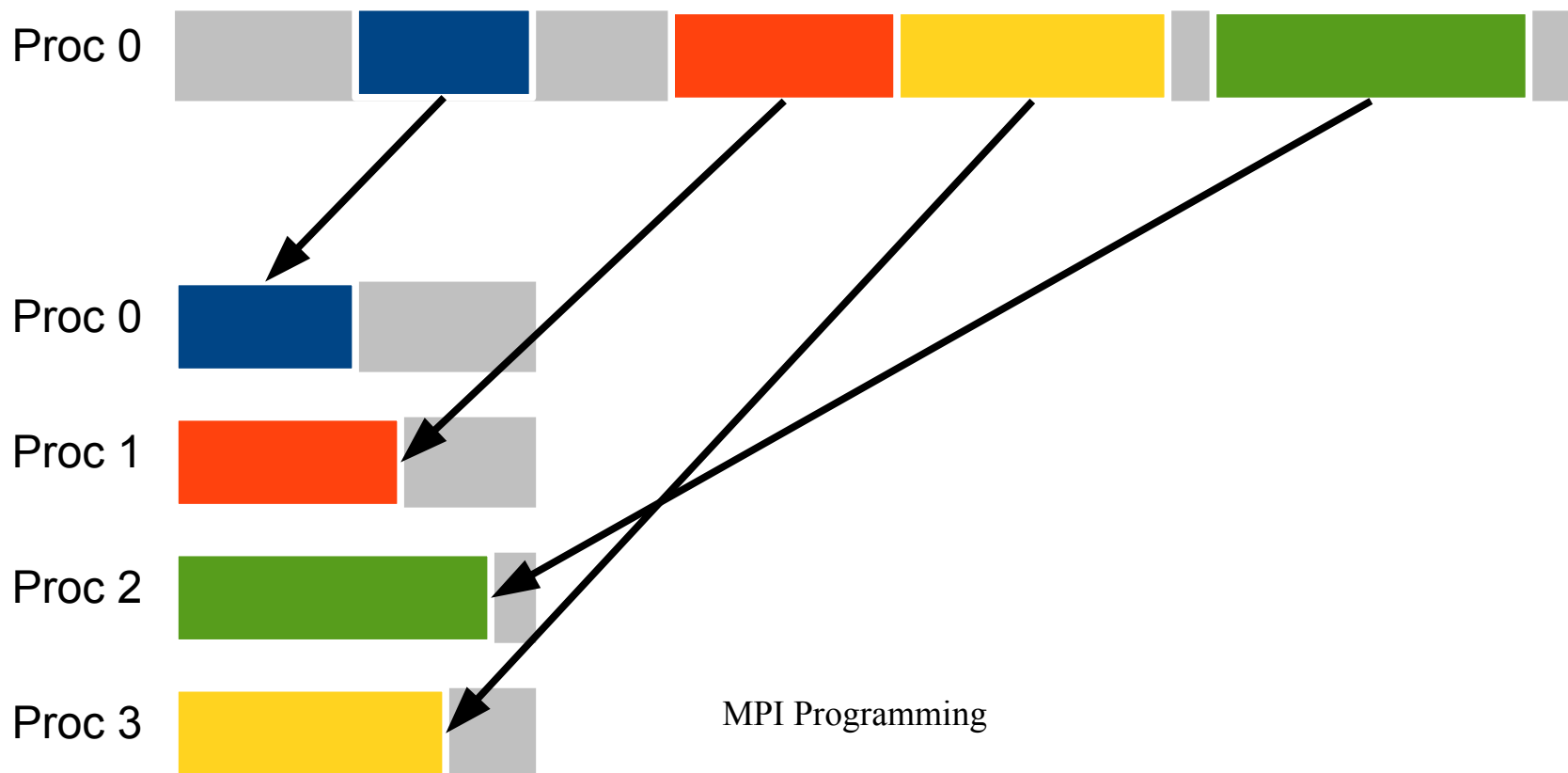
# MPI\_Scatter()

- Contiguous data
- Uniform message size

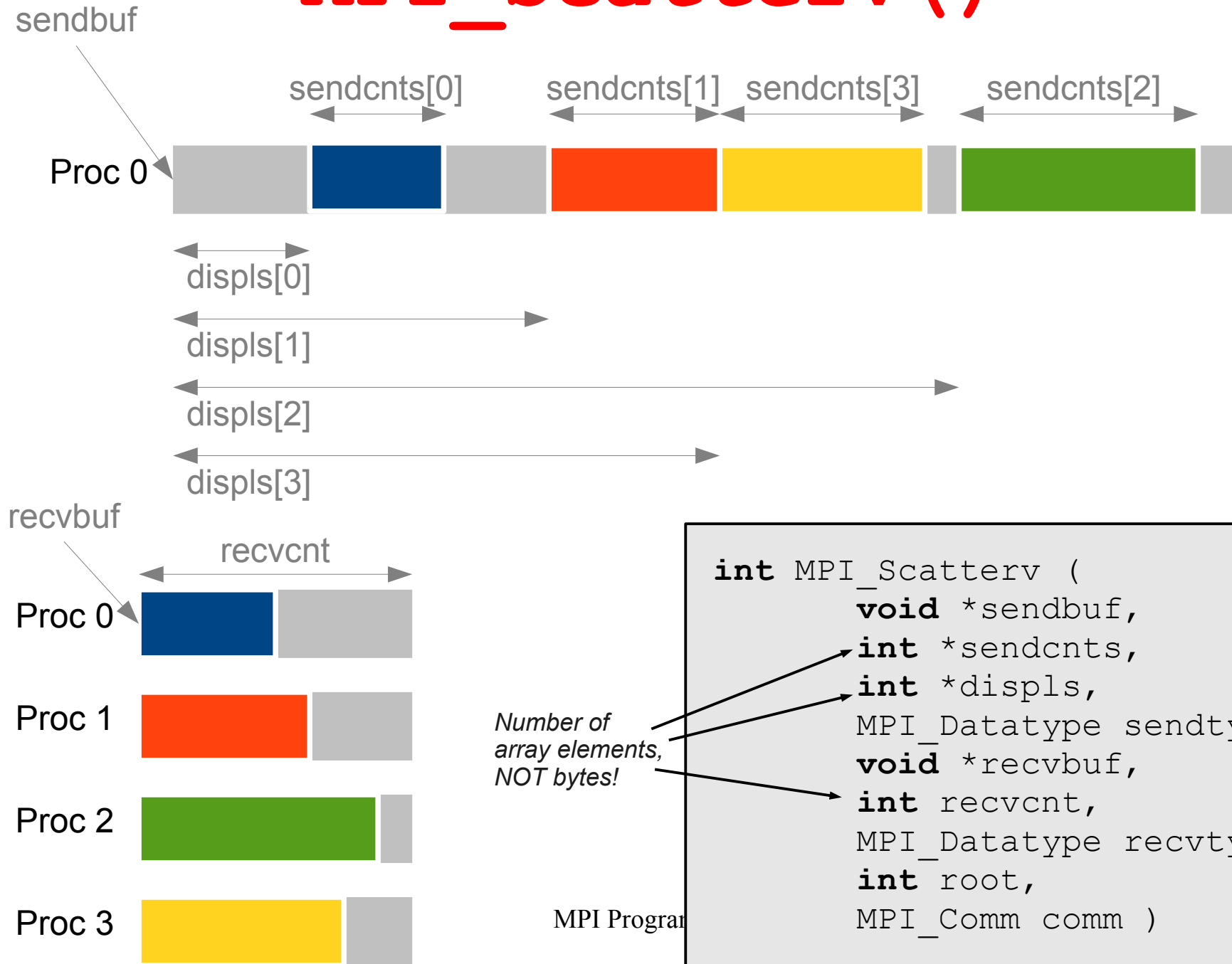


# MPI\_Scatterv() / MPI\_Gatherv()

- Gaps are allowed between messages in source data
- Irregular message sizes are allowed
- Data can be distributed to processes in any order

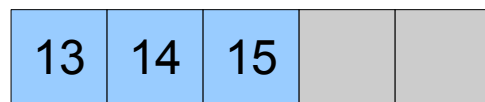


# MPI\_Scatterv()



# Example

```
int sendbuf[] = {10, 11, 12, 13, 14, 15, 16}; /* at master */
int displs[] = {3, 0, 1}; /* assume P=3 MPI processes */
int sendcnts[] = {3, 1, 4};
int recvbuf[5];
...
MPI_Scatterv(sendbuf, sendcnts, displs, MPI_INT, recvbuf, 5,
MPI_INT, 0, MPI_COMM_WORLD);
```



Proc 0



Proc 1

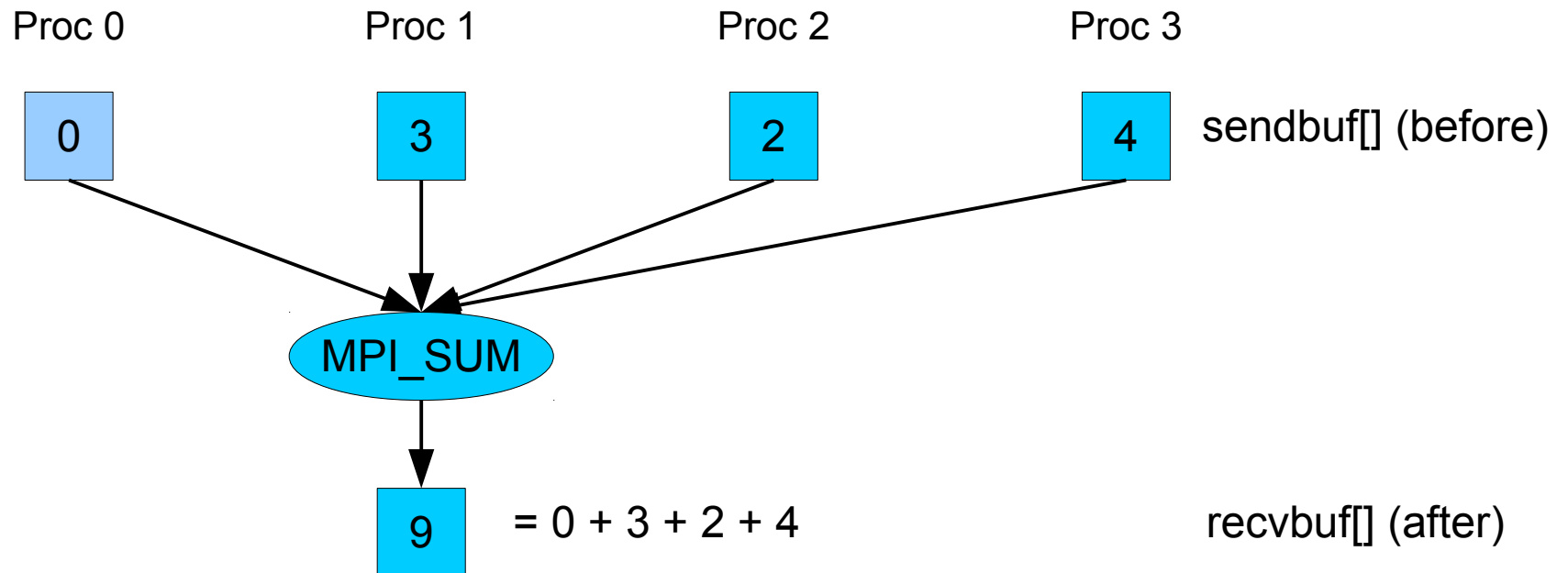


Proc 2

# MPI\_Reduce ()

Performs a reduction and place result in one process

```
count = 1;  
dst = 1; /* result will be placed in process 1 */  
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dst, MPI_COMM_WORLD);
```



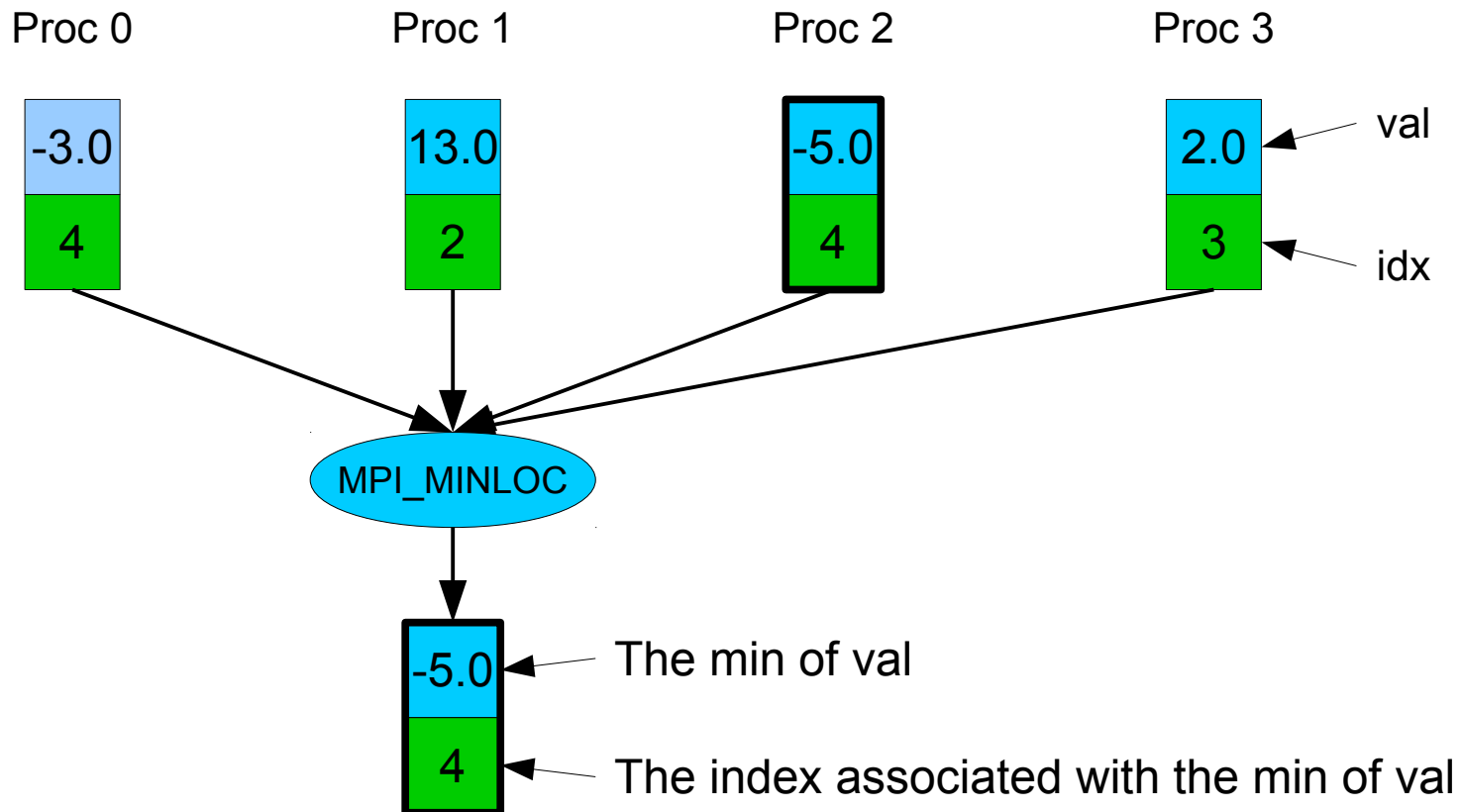
# Predefined reduction operators

Operation Value	Meaning
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical AND
<code>MPI_BAND</code>	Bitwise AND
<code>MPI_LOR</code>	Logical OR
<code>MPI_BOR</code>	Bitwise OR
<code>MPI_LXOR</code>	Logical exclusive OR
<code>MPI_BXOR</code>	Bitwise exclusive OR
<code>MPI_MAXLOC</code>	Maximum and location of maximum
<code>MPI_MINLOC</code>	Minimum and location of minimum

# MINLOC and MAXLOC

Compute global min/max and return an associated index

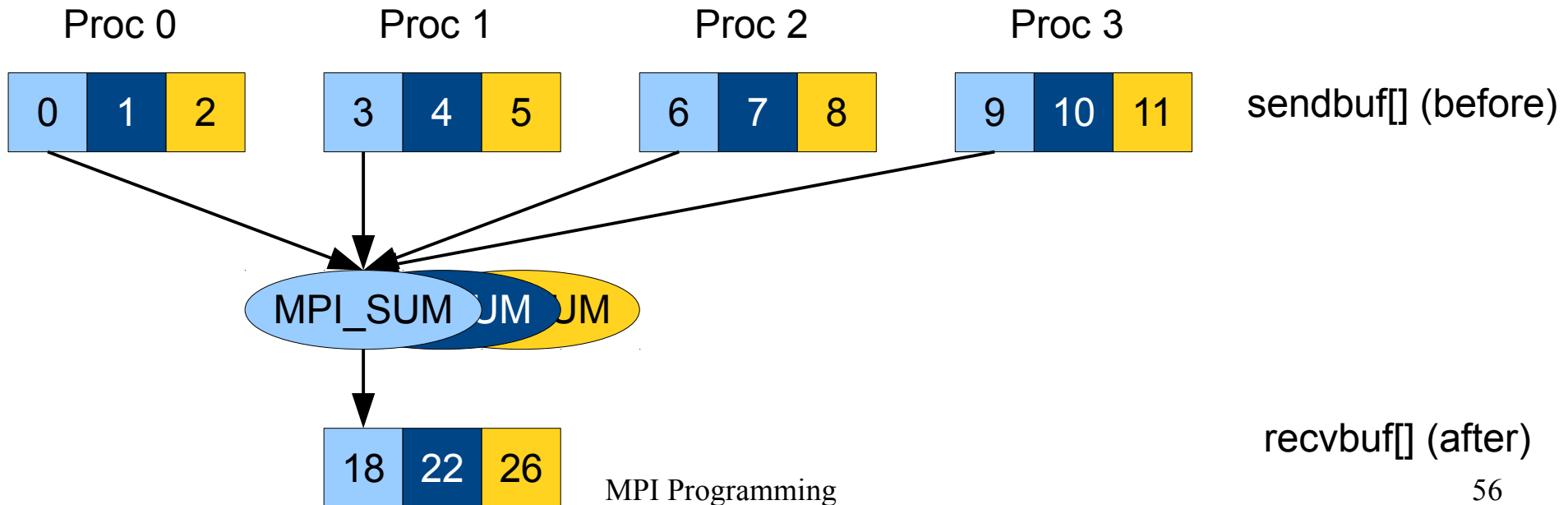
```
struct {double val; int idx} in, out;  
dst = 1; /* result will be placed in process 1 */  
MPI_Reduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MINLOC, dst, MPI_COMM_WORLD);
```



# MPI\_Reduce()

- If  $\text{count} > 1$ , `recvbuf[i]` is the reduction of all elements `sendbuf[i]` at the various processes

```
count = 3;  
dst = 1;  
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dst, MPI_COMM_WORLD);
```





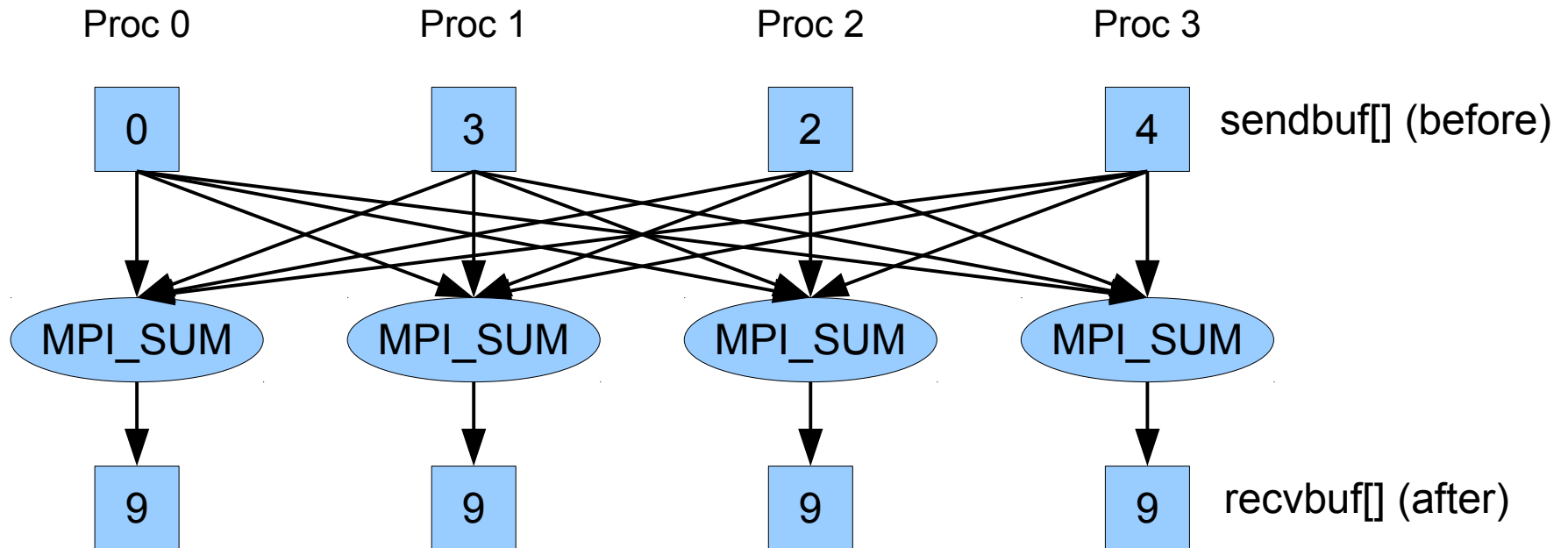
# Parallel trapezoid (with reduction)

- See [mpi-trap1.c](#)

# MPI\_Allreduce()

Performs a reduction and place result in all processes

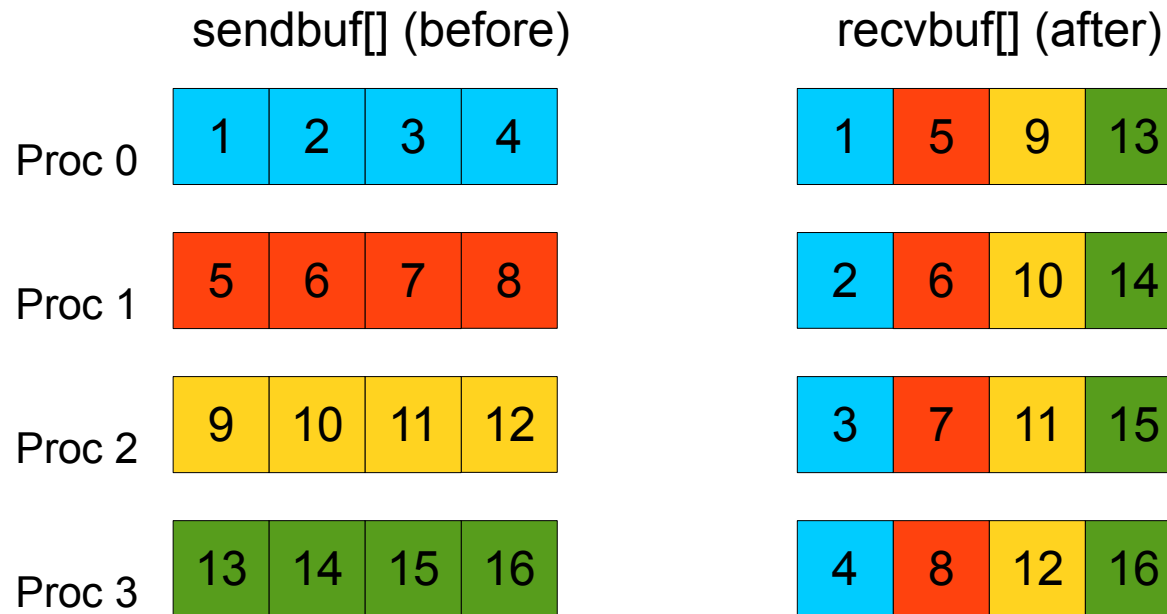
```
count = 1;  
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



# MPI\_Alltoall()

Each process performs a scatter operation

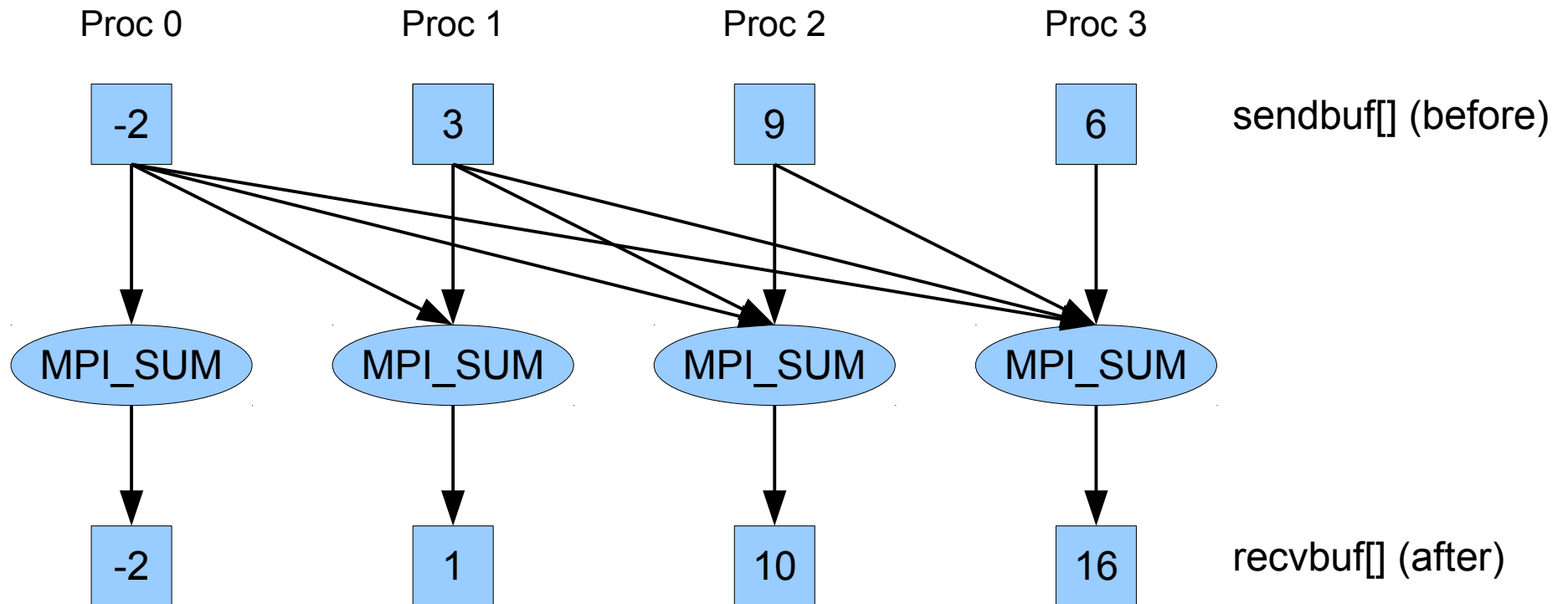
```
sendcnt = 1;  
recvcnt = 1;  
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,  
             recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
```



# MPI\_Scan()

Compute the inclusive scan

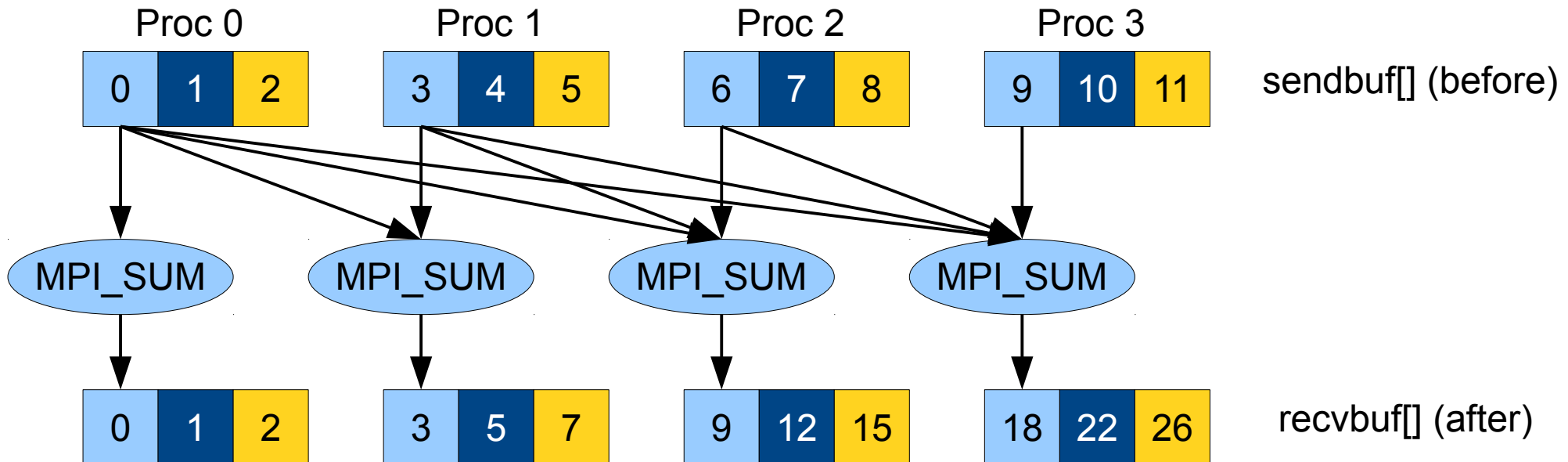
```
count = 1;  
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



# MPI\_Scan()

- If  $\text{count} > 1$ , `recvbuf[i]` at proc.  $j$  is the scan of all elements `sendbuf[i]` at the first  $j$  processes (incl.)

```
count = 3;  
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



# Collective Communication Routines / 1

## MPI\_Barrier(comm)

- Synchronization operation. Creates a barrier synchronization in a group. Each process, when reaching the MPI\_Barrier call, blocks until all processes in the group reach the same MPI\_Barrier call. Then all processes can continue.

## MPI\_Bcast(buffer, count, datatype, root, comm)

- Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

## MPI\_Scatter(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)

## MPI\_Scatterv(sendbuf, sendcnts[], displs[], sendtype, recvbuf, int recvcnt, recvtype, root, comm)

- Distributes distinct messages from a single source process to each process in the group

## MPI\_Gather(sendbuf, sendcnt, sendtype, recvbuf, recvcount, recvtype, root, comm)

## MPI\_Gatherv(sendbuf, sendcnt, sendtype, recvbuf, recvcnts[], displs[], recvtype, root, comm)

- Gathers distinct messages from each process in the group to a single destination process. This routine is the reverse operation of MPI\_Scatter

## MPI\_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- Concatenation of data to all processes in a group. Each process in the group, in effect, performs a one-to-all broadcasting operation within the group

## MPI\_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

- Applies a reduction operation on all processes in the group and places the result in one process

# Collective Communication Routines / 2

## `MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)`

- Collective computation operation + data movement. Does an element-wise reduction on a vector across all processes in the group; all processes receive the result. This is logically equivalent to a `MPI_Reduce()` followed by `MPI_Bcast()`

## `MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)`

- Performs a scan with respect to a reduction operation across a process group.

## `MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm)`

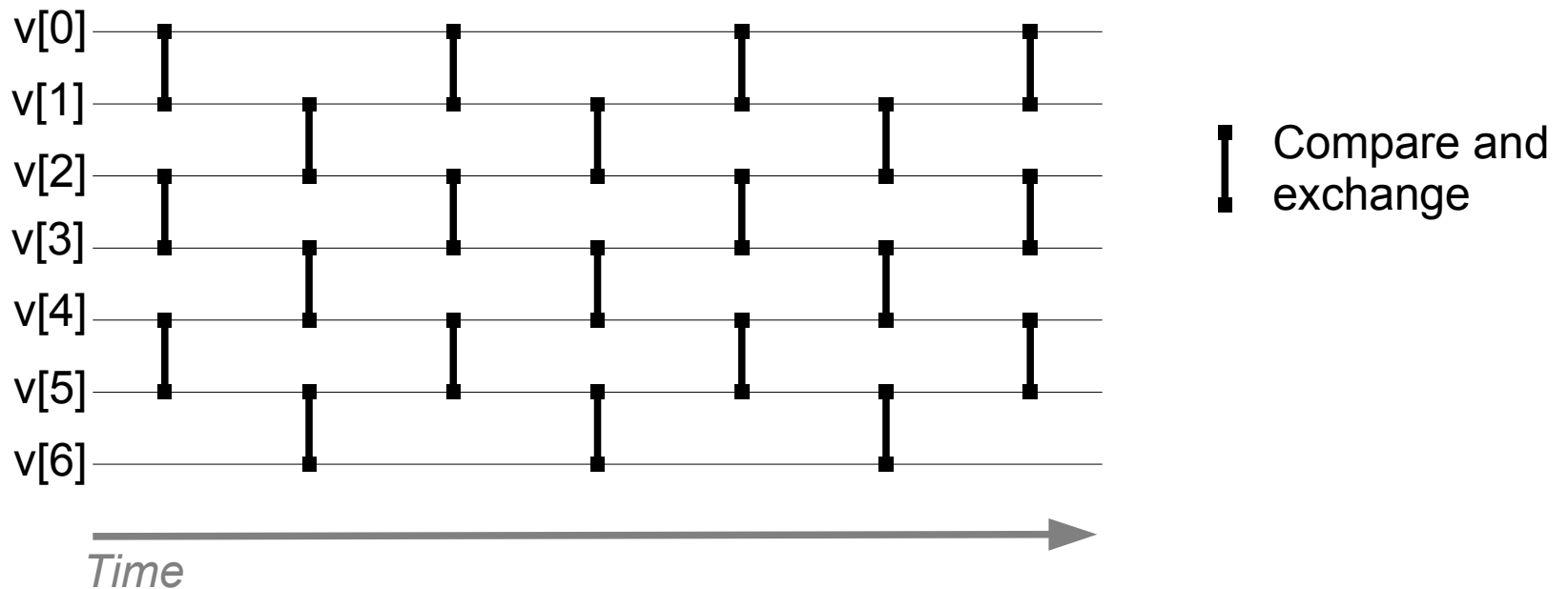
- Data movement operation. Each process in a group performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

# Odd-Even sort with MPI



# Odd-Even Transposition Sort

- Variant of bubble sort
- First, compare all (even, odd) pairs of adjacent elements; exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above



# Odd-Even Transposition Sort

- Variant of bubble sort
- First, compare all (even, odd) pairs of adjacent elements; exchange them if in the wrong order
- Then compare all (odd, even) pairs, exchanging if necessary; repeat the step above

```
for (phase = 0; phase < n; phase++) {
    if (phase % 2 == 0) {
        for (i=0; i<n-1; i += 2) {
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
        }
    } else {
        for (i=1; i<n-1; i += 2) {
            if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
        }
    }
}
```

# Choosing the Granularity

- Communication in distributed memory systems is very expensive, therefore working on individual array elements does not make sense
- We achieve a coarser granularity by splitting the array in blocks and applying odd-even sort at the block level

G. Baudet and D. Stevenson, "*Optimal Sorting Algorithms for Parallel Computers*," in IEEE Transactions on Computers, vol. C-27, no. 1, pp. 84-87, Jan. 1978. doi:10.1109/TC.1978.1674957

# Odd-Even Transposition Sort

- The vector is split in blocks that are assigned to MPI processes
- Each process sorts its block (e.g., using `qsort`)
- At each exchange-swap step:
  - Process  $i$  sends a copy of its chunk to process  $i + 1$
  - Process  $i + 1$  sends a copy of its chunk to process  $i$
  - Process  $i$  merges the chunks, discards **upper** half
  - Process  $i + 1$  merges the chunks, discards **lower** half

# Exchange-Merge step

Process  $i$

Process  $i + 1$

10 7 12 33 9

13 52 14 97 31

sort

sort

7 9 10 12 33

13 14 31 52 97

Send to partner

7 9 10 12 33 13 14 31 52 97

7 9 10 12 33 13 14 31 52 97

merge

merge

7 9 10 12 13 14 31 33 52 97

7 9 10 12 13 14 31 33 52 97

discard upper half

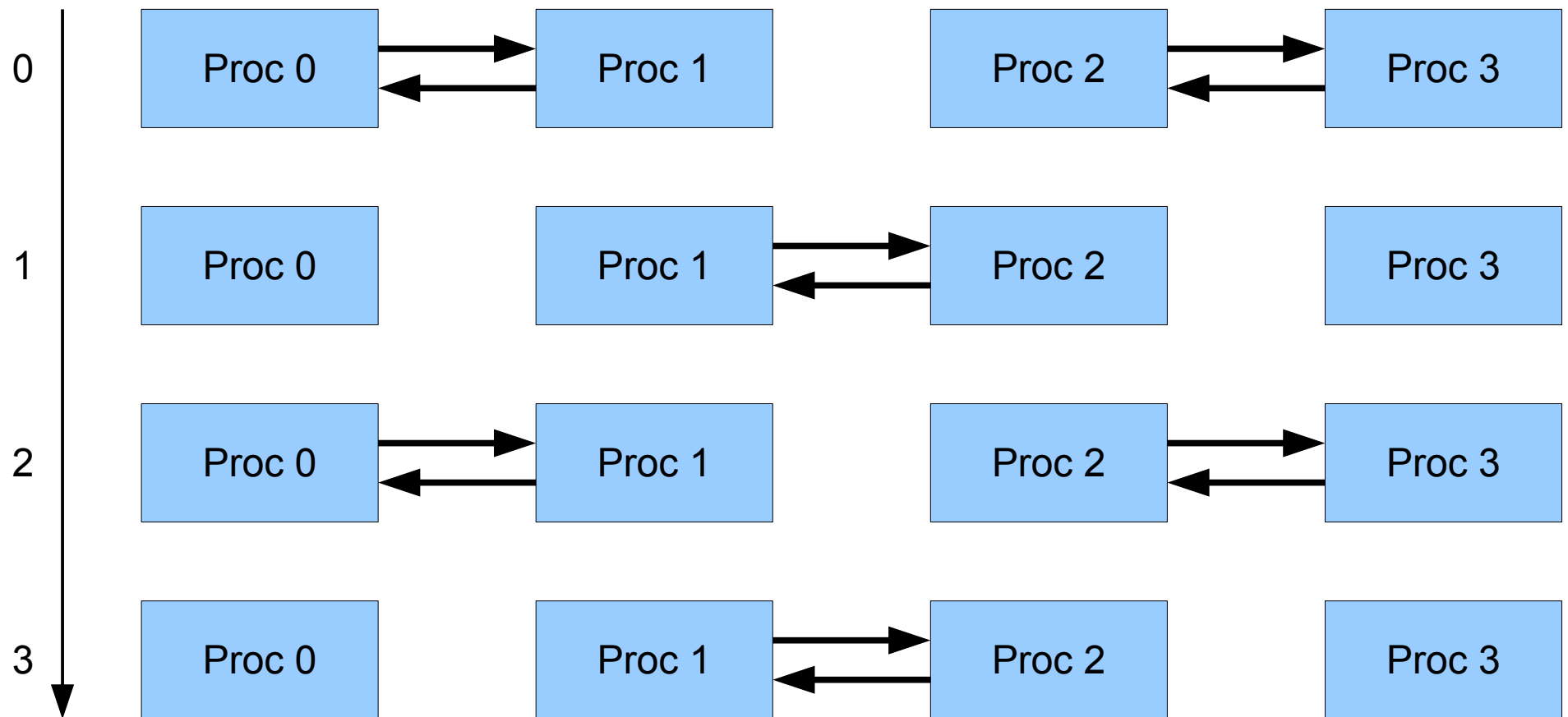
discard lower half

7 9 10 12 13

14 31 33 52 97

# Communication pattern for odd-even sort

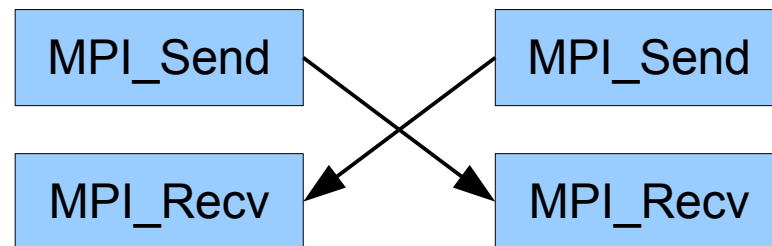
phase



# Beware of the (possible) deadlock!!

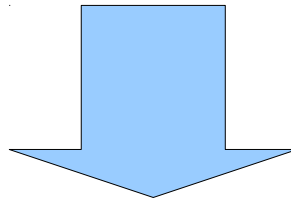
```
Sort local values
for (phase=0; phase < comm_sz; phase++) {
    partner = compute_partner(phase, my_rank);
    if (I am not idle) {
    → Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner) {
          Keep smaller keys;
      } else {
          Keep larger keys;
      }
    }
}
```

*Depending on the MPI implementation, the send operation may block if there is no matching receive at the other end; unfortunately, all receive are executed only after the send completes!*



# Solution 1 (ugly): restructure communications

```
MPI_Send(msg1, size, MPI_INT, partner, 0, comm);  
MPI_Recv(msg2, size, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);
```



```
if ( my_rank % 2 == 0 ) {  
    MPI_Send(msg1, size, MPI_INT, partner, 0, comm);  
    MPI_Recv(msg2, size, MPI_INT, partner,  
             0, comm, MPI_STATUS_IGNORE);  
} else {  
    MPI_Recv(msg2, size, MPI_INT, partner,  
             0, comm, MPI_STATUS_IGNORE);  
    MPI_Send(msg1, size, MPI_INT, partner, 0, comm);  
}
```



# Better: use `MPI_Sendrecv()`

- Executes a blocking send and a receive in a single call
  - *dest* and the *source* can be the same or different
  - MPI schedules the communications so that the program won't hang or crash
- `MPI_Sendrecv()` can be matched by `MPI_Send()` / `MPI_Recv()`
  - However, it is **very unlikely** that you will ever need to do that

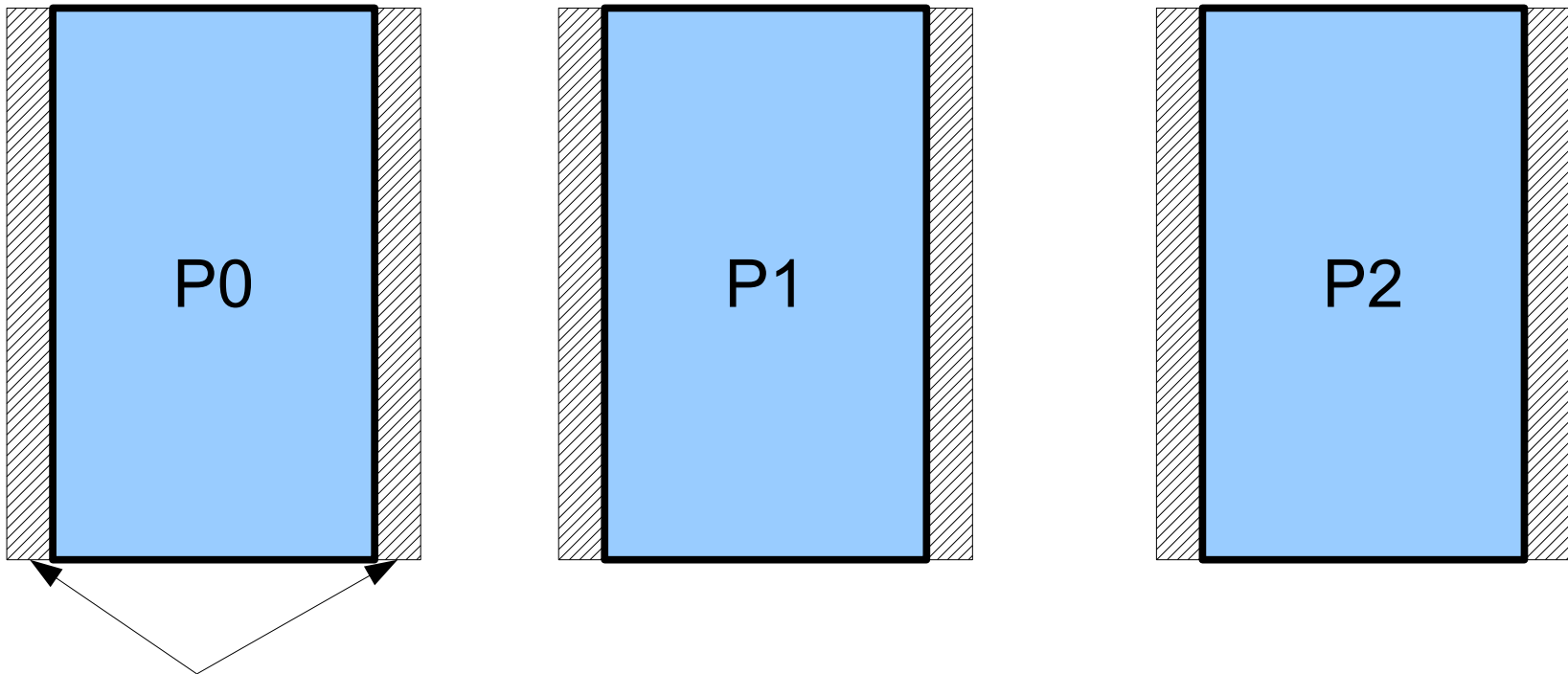
```
int MPI_Sendrecv(  
    void*          sendbuf,  
    int           sendcount,  
    MPI_Datatype  sendtype,  
    int           dest,  
    int           sendtag,  
    void*          recvbuf,  
    int           recvcount,  
    MPI_Datatype  recvtype,  
    int           source,  
    int           recvtag,  
    MPI_Comm      comm,  
    MPI_Status*   status  
)
```

See [mpi-odd-even.c](#)

# MPI Datatypes

# Example

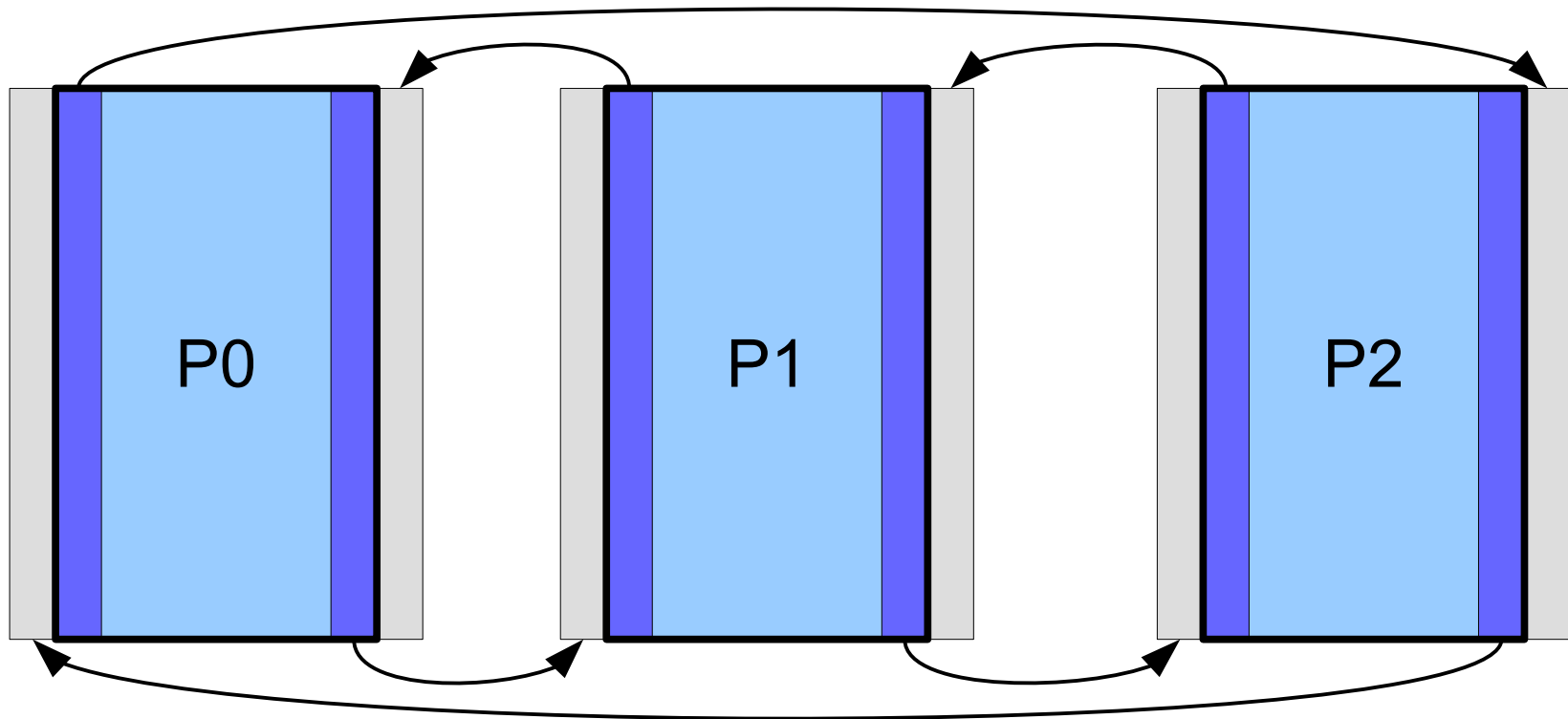
- Let us consider a two-dimensional domain
- (\*, Block) decomposition
  - with ghost cells along the vertical edges only



Ghost cells

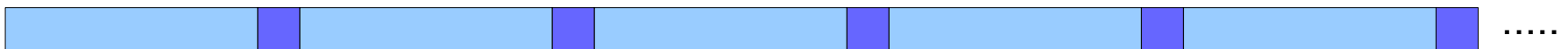
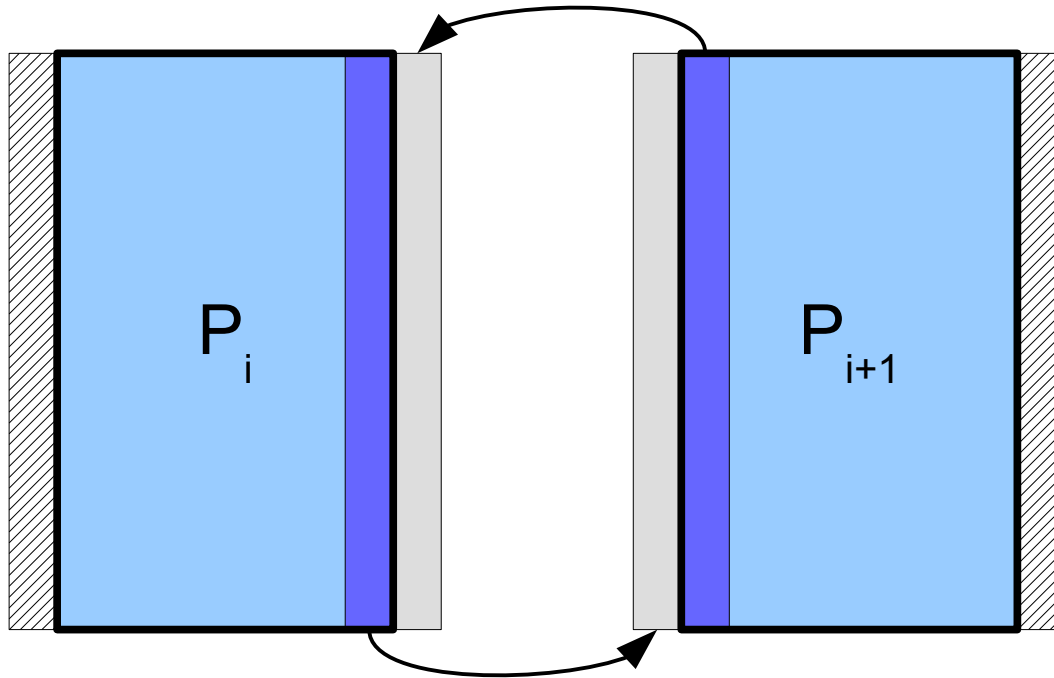
# Example

- At each step, nodes must exchange their outer columns with neighbors



# Example

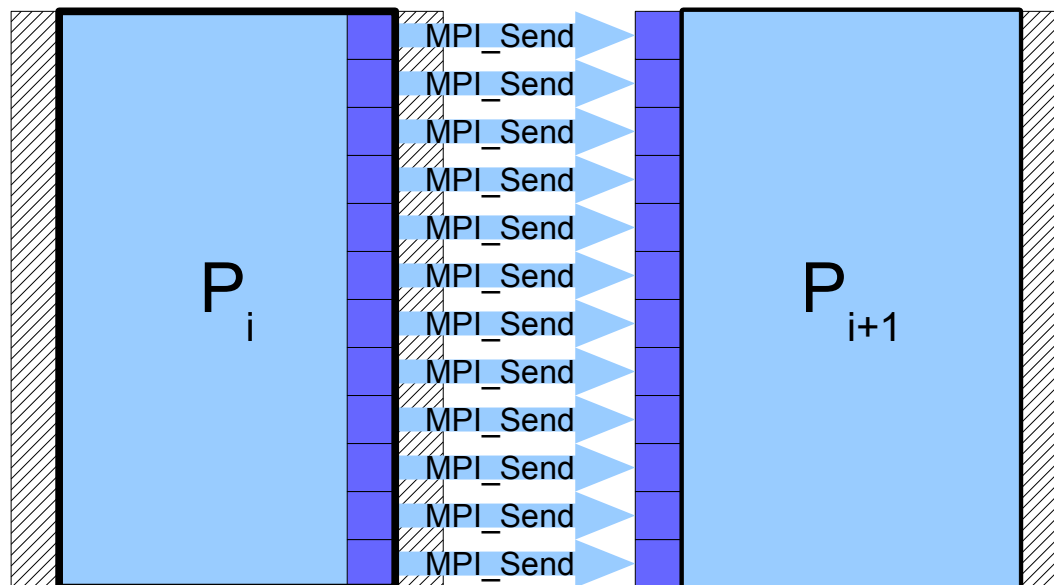
- In the C language, matrices are stored row-wise
  - Elements of the same column are not contiguous in memory





# Example

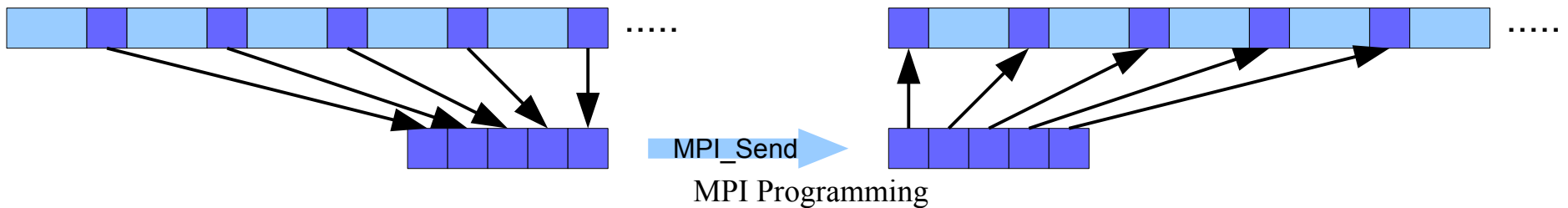
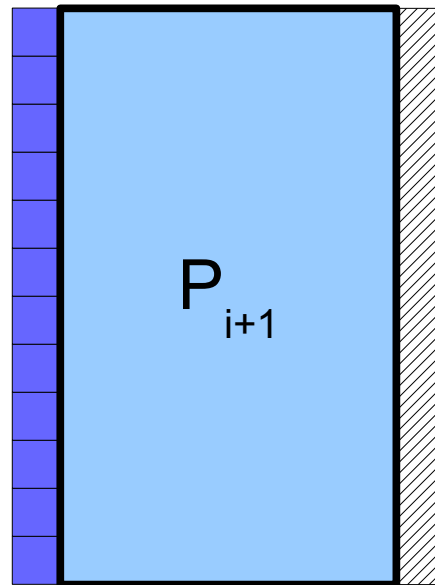
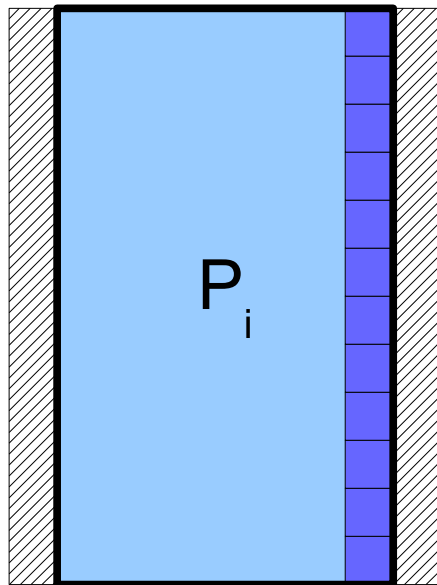
- **The BAD** solution: send each element with `MPI_Send` (or `MPI_Isend`)





# Example

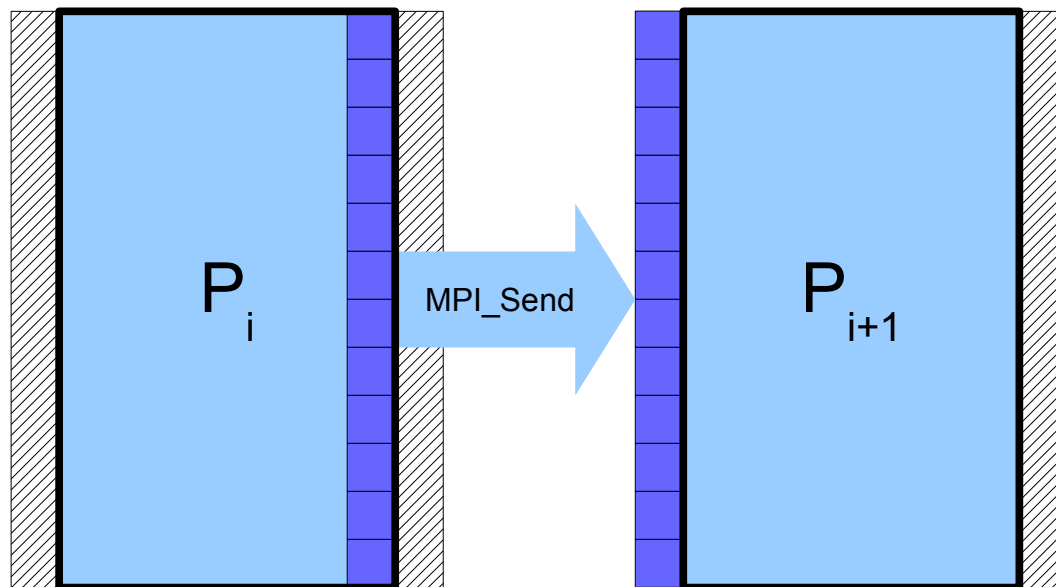
- **The UGLY** solution: copy the column into a temporary buffer; `MPI_Send()` the buffer; fill the destination column





# Example

- **The GOOD** solution: define a new datatype for the column, and **MPI\_Send** the column directly





# MPI Derived Datatypes

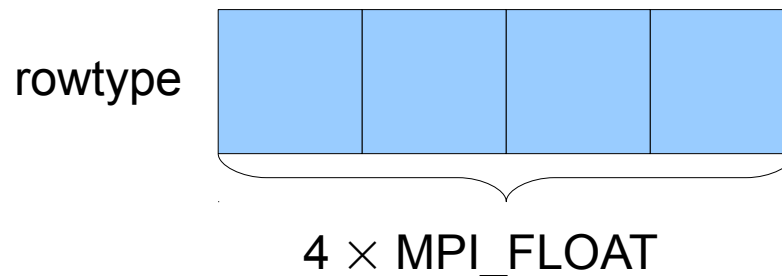
- MPI provides several methods for constructing derived data types:
  - **Contiguous**: a contiguous block of elements
  - **Vector**: a strided vector of elements
  - **Indexed**: an irregularly spaced set of blocks of the **same** type
  - **Struct**: an irregularly spaced set of blocks possibly of **different** types
- Other functions
  - `MPI_Type_commit(...)` commits a new datatype
  - `MPI_Type_free(...)` deallocates a datatype object

# MPI\_Type\_contiguous()

```
MPI_Datatype rowtype;  
MPI_Type_contiguous( 4, MPI_FLOAT, &rowtype );  
MPI_Type_commit(&rowtype);
```

`int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)`

- A contiguous block of `count` elements of an existing MPI type `oldtype`
  - `oldtype` can be another previously defined custom datatype



# MPI\_Type\_contiguous()

```
int count = 4
MPI_Datatype rowtype;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
```

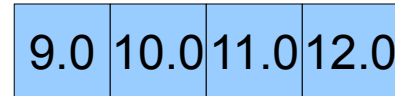


See [mpi-type-contiguous.c](#)

	1.0	2.0	3.0	4.0	
	5.0	6.0	7.0	8.0	
a[2][0]	9.0	10.0	11.0	12.0	a[4][4]
	13.0	14.0	15.0	16.0	

The table shows a 4x4 grid of floating-point numbers. The third row (9.0, 10.0, 11.0, 12.0) is highlighted in light blue. An arrow points from the label 'a[2][0]' to the first cell of this row. The label 'a[4][4]' is positioned to the right of the grid.

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, MPI_COMM_WORLD);
```



1 element of type rowtype

# MPI\_Type\_vector()

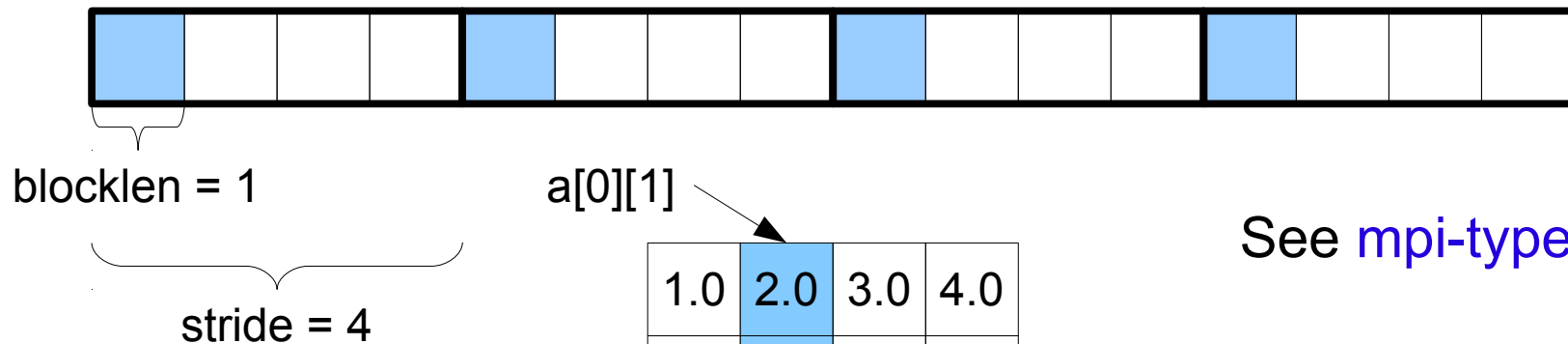
```
MPI_Datatype columntype;  
MPI_Type_vector( 4, 1, 4, MPI_FLOAT, &columntype );  
MPI_Type_commit(&columntype);
```

`int MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)`

- A regularly spaced array of elements of the existing MPI type `oldtype`
  - `count` number of blocks
  - `blocklen` number of elements of each block
  - `stride` number of elements between start of contiguous blocks
  - `oldtype` can be another previously defined datatype

# MPI\_Type\_vector()

```
int count = 4, blocklen = 1, stride = 4;  
MPI_Datatype columntype;  
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &columntype);  
MPI_Type_commit(&columntype);
```



a[0][1]

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

See [mpi-type-vector.c](#)

a[4][4]

```
MPI_Send(&a[0][1], 1, columntype, dest, tag, MPI_COMM_WORLD);
```

2.0	6.0	10.0	14.0
-----	-----	------	------

1 element of type columntype

# Quiz

```
int count = 4, blocklen = 2, stride = 4;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

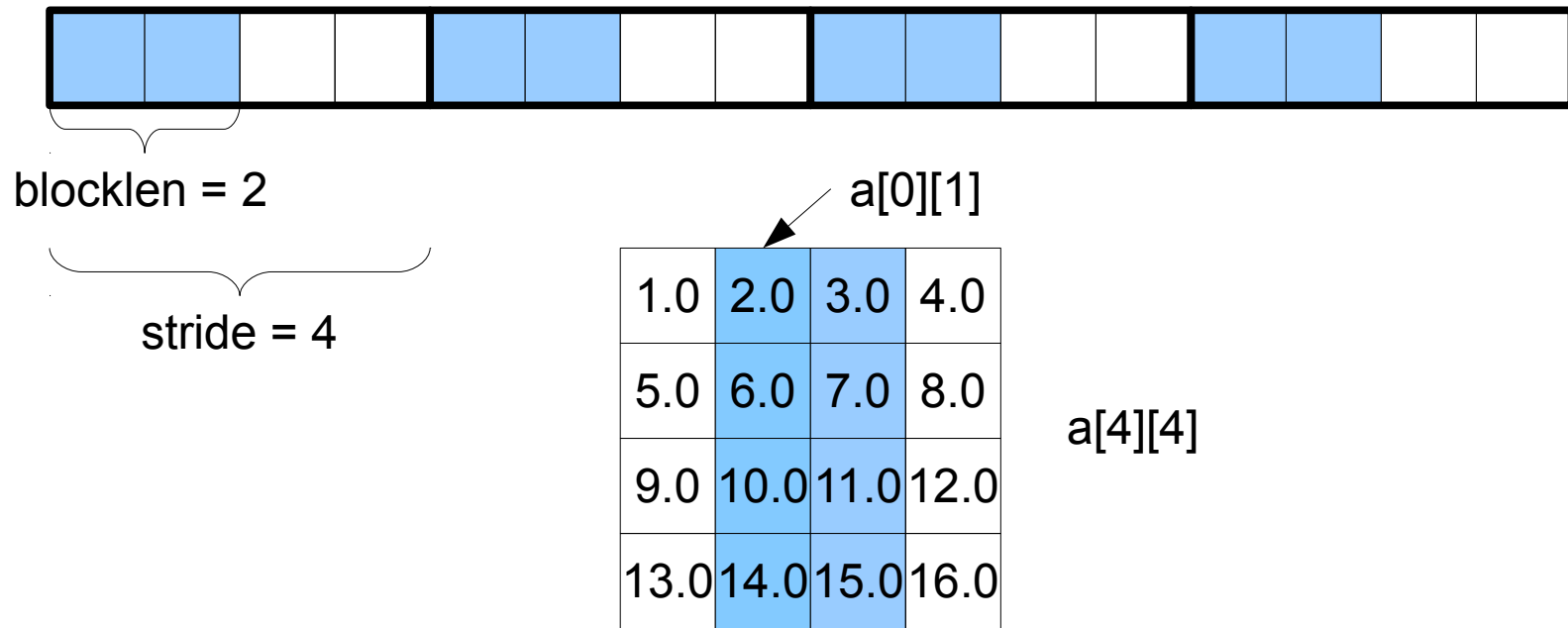
a[4][4]

```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

**Which data are being transmitted?**

# Quiz

```
int count = 4, blocklen = 2, stride = 4;  
MPI_Datatype newtype;  
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

2.0 3.0 6.0 7.0 10.0 11.0 14.0 15.0

# Quiz

```
int count = 3, blocklen = 1, stride = 5;
MPI_Datatype newtype;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

**Which data are being transmitted?**



# Quiz

```
int count = 3, blocklen = 1, stride = 5;  
MPI_Datatype newtype;  
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```



blocklen = 1

stride = 5

a[0][1]

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

2.0	7.0	12.0
-----	-----	------

# Quiz

```
int count = ???, blocklen = ???, stride = ???;  
MPI_Datatype newtype;  
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```

Fill the ??? with the parameters required to get the behavior below

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

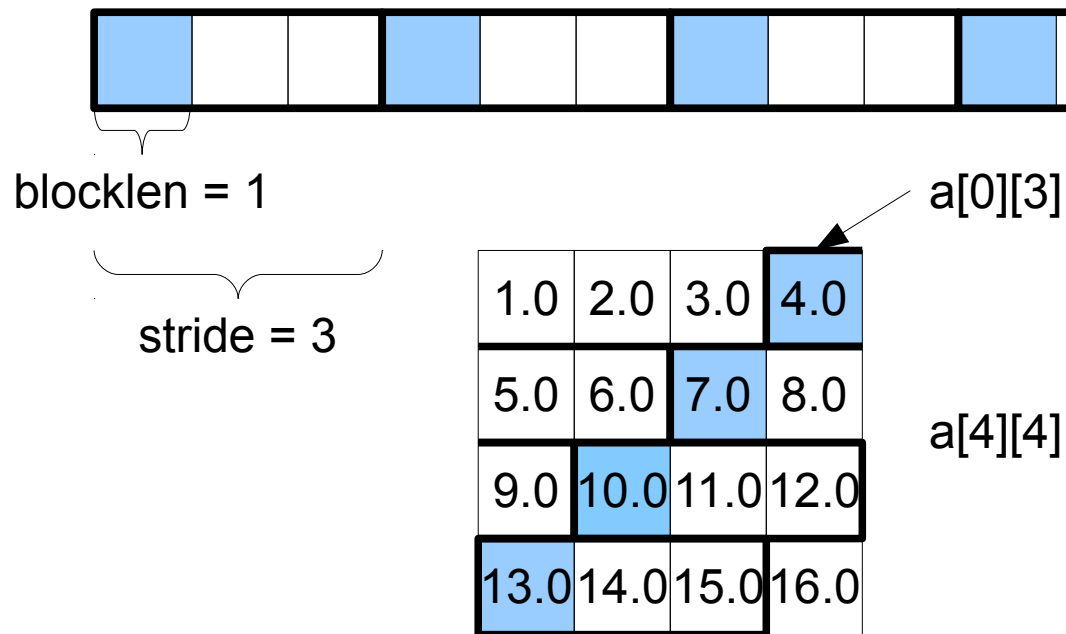
a[4][4]

```
MPI_Send(???????, 1, newtype, dest, tag, MPI_COMM_WORLD);
```

4.0	7.0	10.0	13.0
-----	-----	------	------

# Quiz

```
int count = 4; blocklen = 1, stride = 3;  
MPI_Datatype newtype;  
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0][3], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

4.0	7.0	10.0	13.0
-----	-----	------	------

# MPI\_Type\_indexed()

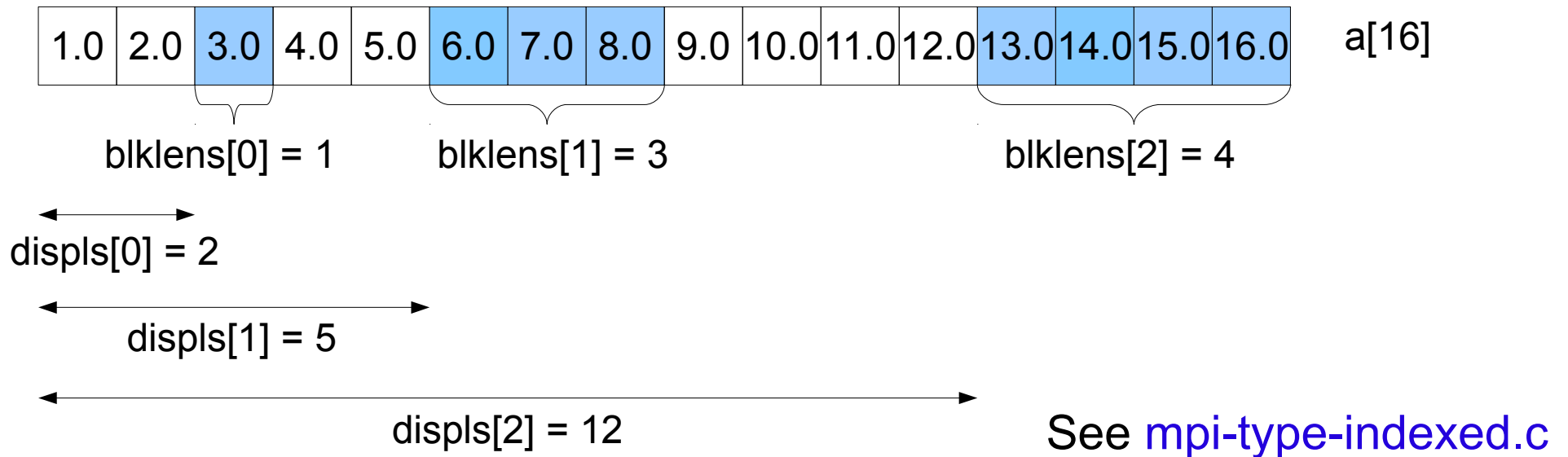
```
MPI_Datatype newtype;  
MPI_Type_indexed( ... );  
MPI_Type_commit(&newtype);
```

```
int MPI_Type_indexed(int count, const int  
array_of_blklen[], const int array_of_displ[],  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

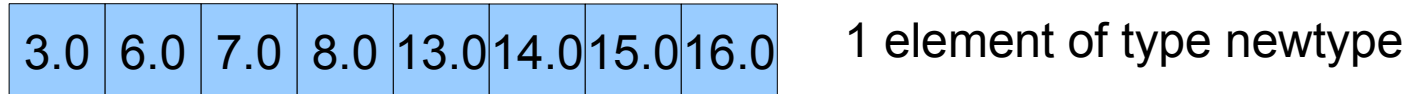
- An irregularly spaced set of blocks of elements of an existing MPI type `oldtype`
  - `count` number of blocks
  - `array_of_blklen` number of elements in each block
  - `array_of_displ` displacement of each block with respect to the beginning of the data structure
  - `oldtype` can be another previously defined datatype

# MPI\_Type\_indexed()

```
int count = 3; int blkLens[] = {1, 3, 4}; int displs[] = {2, 5, 12};
MPI_Datatype newtype;
MPI_Type_indexed(count, blkLens, displs, MPI_FLOAT, &newtype);
MPI_Type_commit(&newtype);
```



```
MPI_Send(&a[0], 1, newtype, dest, tag, MPI_COMM_WORLD);
```



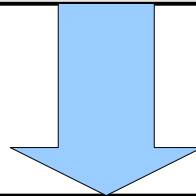
# MPI\_Type\_indexed()

```
int count = 3; int blkLens[] = {1, 3, 4}; int displs[] = {2, 5, 12};  
MPI_Datatype newtype;  
MPI_Type_indexed(count, blkLens, displs, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0
-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------

 a[16]

```
MPI_Send(&a[0], 1, newtype, dest, tag, MPI_COMM_WORLD);
```



```
MPI_Recv(&b[0], 1, newtype, src, tag, MPI_COMM_WORLD);
```

		3.0			6.0	7.0	8.0					13.0	14.0	15.0	16.0
--	--	-----	--	--	-----	-----	-----	--	--	--	--	------	------	------	------

 b[16]

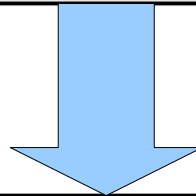
# MPI\_Type\_indexed()

```
int count = 3; int blkLens[] = {1, 3, 4}; int displs[] = {2, 5, 12};  
MPI_Datatype newtype;  
MPI_Type_indexed(count, blkLens, displs, MPI_FLOAT, &newtype);  
MPI_Type_commit(&newtype);
```

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0	16.0
-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------

 a[16]

```
MPI_Send(&a[0], 1, newtype, dest, tag, MPI_COMM_WORLD);
```



```
MPI_Recv(&b[0], 8, MPI_FLOAT, src, tag, MPI_COMM_WORLD);
```

3.0	6.0	7.0	8.0	13.0	14.0	15.0	16.0								
-----	-----	-----	-----	------	------	------	------	--	--	--	--	--	--	--	--

 b[16]

# Combining custom datatypes

- The `oldtype` parameter of functions `MPI_Type_contiguous()`, `MPI_Type_vector()` and `MPI_Type_indexed()` can be another user-defined datatype

```
int count, blocklen, stride;
MPI_Datatype vec, vecvec;

count = 2; blocklen = 2; stride = 3;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &vec);
MPI_Type_commit(&vec);
count = 2; blocklen = 1; stride = 3;
MPI_Type_vector(count, blocklen, stride, vec, &vecvec);
MPI_Type_commit(&vecvec);
```

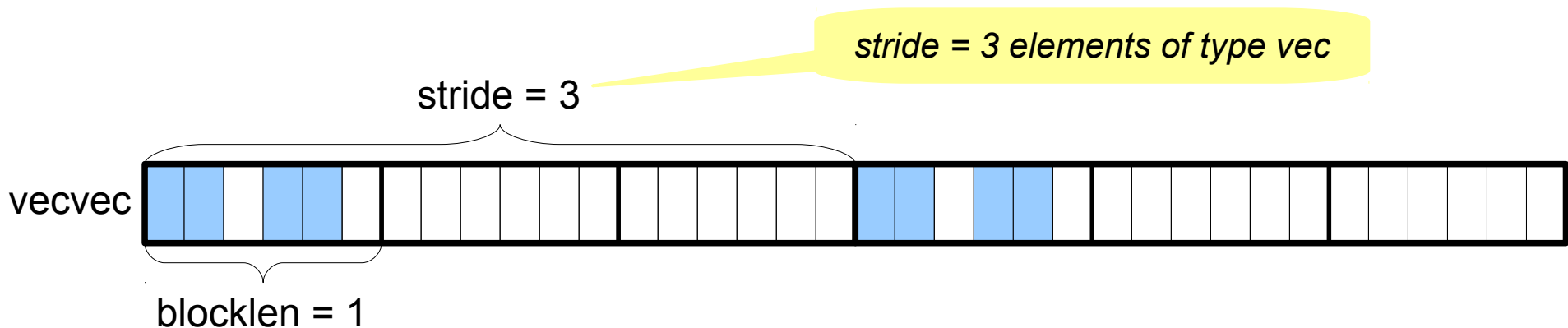
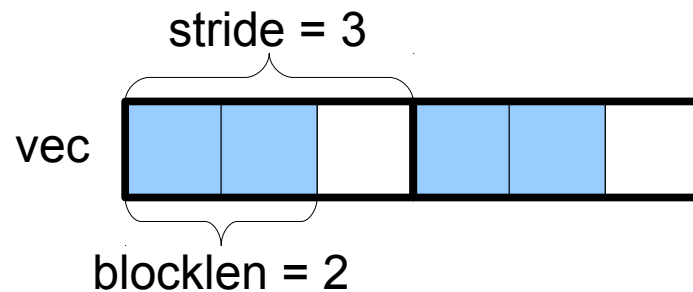


```

int count, blocklen, stride;
MPI_Datatype vec, vecvec;

count = 2; blocklen = 2; stride = 3;
MPI_Type_vector(count, blocklen, stride, MPI_FLOAT, &vec);
MPI_Type_commit(&vec);
count = 2; blocklen = 1; stride = 3;
MPI_Type_vector(count, blocklen, stride, vec, &vecvec);
MPI_Type_commit(&vecvec);

```



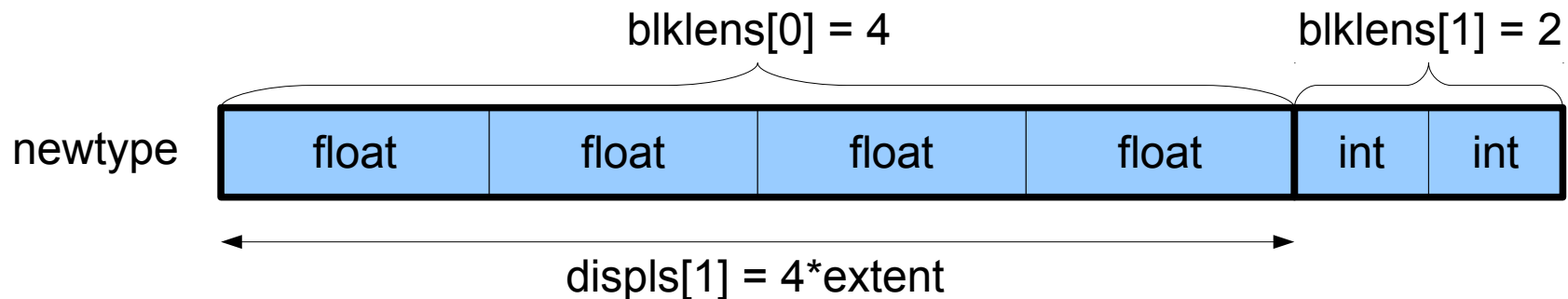
# MPI\_Type\_struct()

```
int MPI_Type_struct(int count, int *array_of_blklen,  
MPI_Aint *array_of_displ, MPI_Datatype  
*array_of_types, MPI_Datatype *newtype)
```

- An irregularly spaced set of blocks of elements of existing MPI types `array_of_types`
  - `count` number of blocks (and number of elements of the arrays `array_of_*`)
  - `array_of_blklen` number of elements in each block
  - `array_of_displ` displacement **in bytes** of each block with respect to the beginning of the data structure (of type `MPI_Aint`)
  - `array_of_types` array of `MPI_Datatype`

# MPI\_Type\_struct()

```
typedef struct {  
    float x, y, z, v;  
    int n, t;  
} particle_t;  
  
int count = 2; int blkLens[] = {4, 2};  
MPI_Aint displs[2], lb, extent;  
MPI_Datatype oldtypes[2] = {MPI_FLOAT, MPI_INT}, newtype;  
MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);  
displs[0] = 0; displs[1] = 4*extent;  
MPI_Type_struct(count, blkLens, displs, oldtypes, &newtype);  
MPI_Type_commit(&newtype);
```



# Concluding Remarks

- Message passing is a very simple model
  - Extremely low level; heavy weight
  - Expense comes from communication and lots of local code
  - Communication code is often more than half
  - Tough to make adaptable and flexible
  - Tough to get right
- Programming model of choice for scalability
  - Widespread adoption due to portability