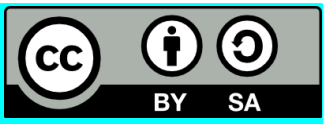
Distributed Memory Programming with MPI

Pacheco chapter 3

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

moreno.marzolla@unibo.it



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://hpc-tutorials.llnl.gov/mpi/ (highly recommended!!)

Introduction

Message Passing and MPI

- Message passing is 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) paradigm
  - 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_rank == 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 Open MPI
  - 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
  - If you have not already done so, generate a public/private key pair on your local machine with **`ssh-keygen`**; do not enter a passphrase
  - Copy the public key to the remote machine:

    `ssh-copy-id` ***`remote-user@remote-host`***

Finding Out About the Environment

- Two important questions that arise frequently in a parallel program are:
  - How many processes are taking part 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!

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

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

MPI

- The following six functions suffice for simple programs:
  - **`MPI_Init`**
  - **`MPI_Finalize`**
  - **`MPI_Comm_size`** (how many processes are there?)
  - **`MPI_Comm_rank`** (who am I?)
  - **`MPI_Send`** (blocking send)
  - **`MPI_Recv`** (blocking receive)
  - **`MPI_Abort`** (aborting the computation)

A Simple MPI Program

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

    /* process 0 sends and process 1 receives */
    if (my_rank == 0) {
        buf = 123456;
        MPI_Send(&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (my_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 processes are organized
  - 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

- Data sent or received is described by a triple (address, count, datatype)
- An MPI datatype is recursively defined as:
  - a predefined type (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 default datatypes

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed lont 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	

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 tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag
  - If the receiver gets a message with a different tag than the one specified in the **`MPI_Recv()`** call, the message is kept on hold and will be matched by a future **`MPI_Recv()`** with the correct tag

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*)
- dest is the rank of the target process
  - If dest is **`MPI_PROC_NULL`**, che **`MPI_Send`** operation has no effect
- tag is the message tag (we will always use 0)
- 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.

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, 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 items is OK, but receiving more is an error (see later)
- status contains further information (e.g., message size), use **MPI_STATUS_IGNORE** if no information is required

Send/Receive message sizes Examples

- Sender sends 10, receiver expects (at most) 10 → OK
- Sender sends 10, receiver expects (at most) 20 → OK
- Sender sends 20, receiver expects (at most) 10 → MPI_ERR_TRUNCATE
- **`MPI_Recv()`** ensures that the receiver always gets the full message that was sent
  - If 10 items are sent, a matching **`MPI_Recv()`** will return when all 10 items have been received

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**

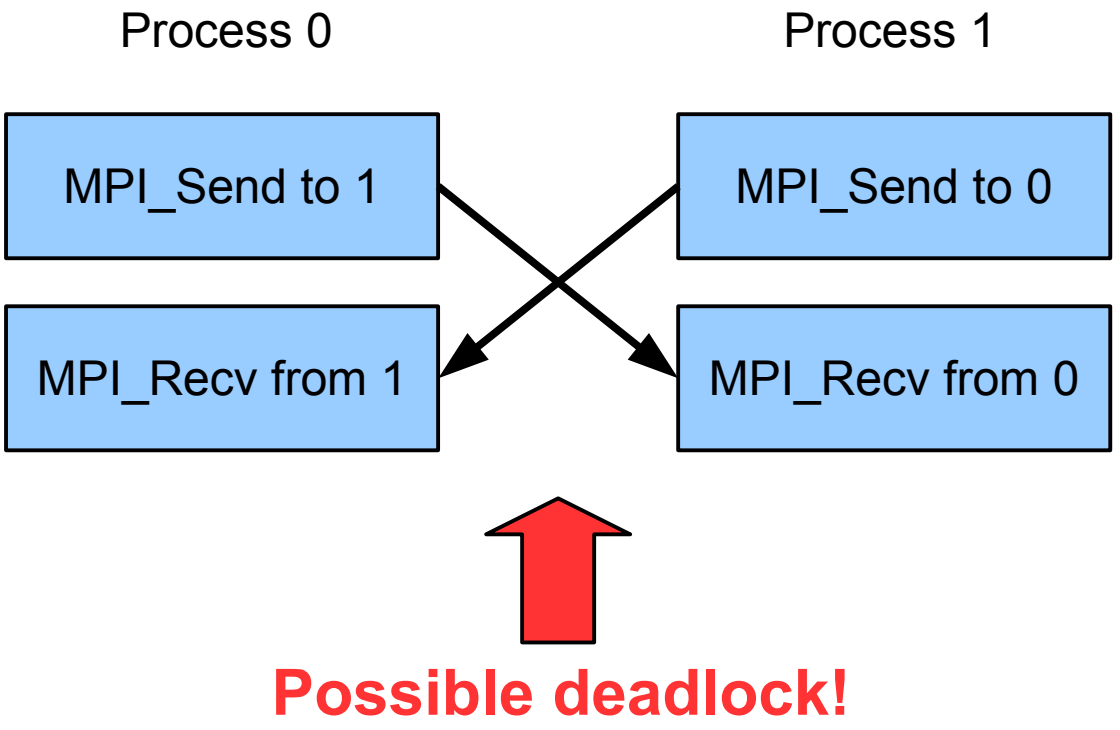
MPI_Get_count()

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

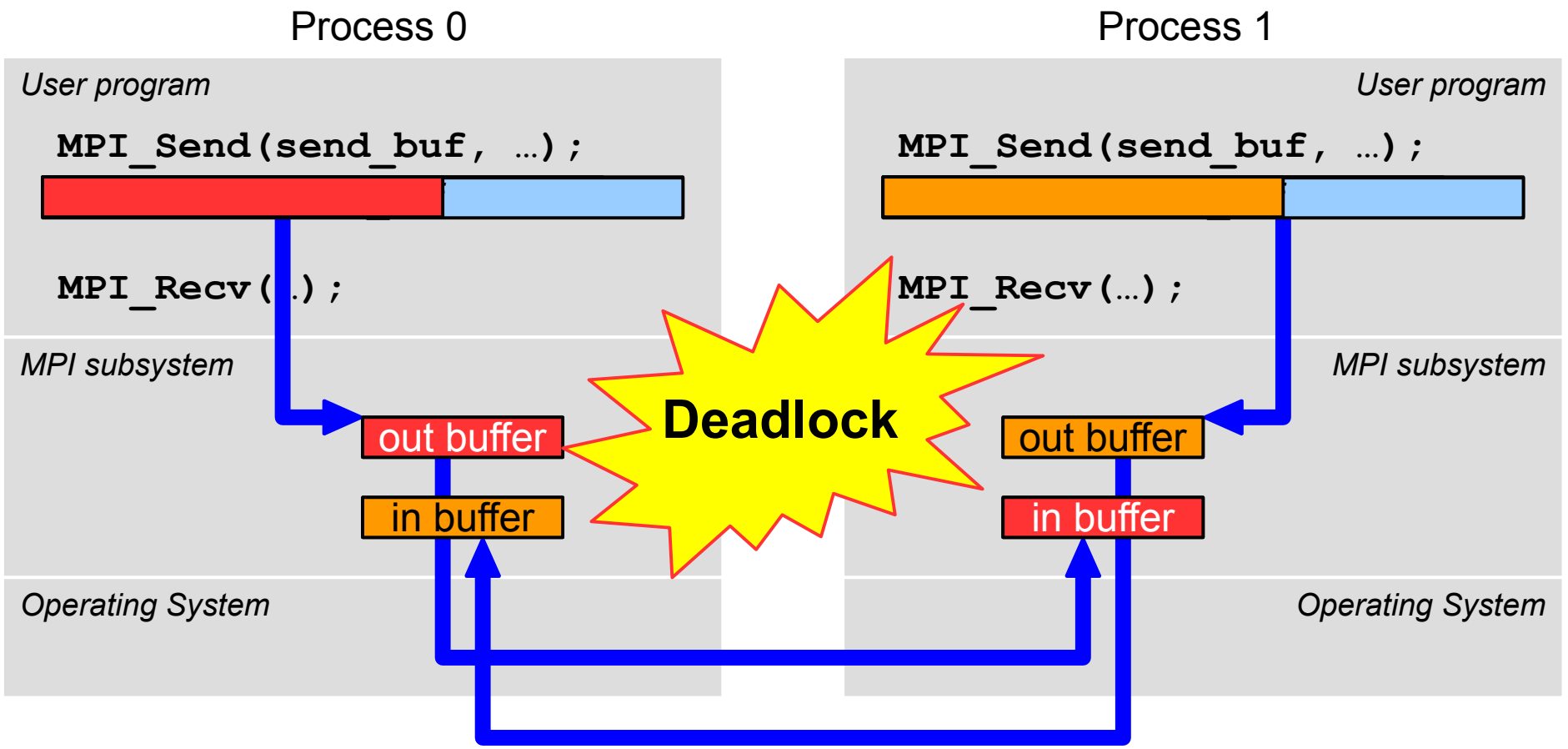
- **MPI_Recv** may complete even if less than *count* elements have been received
  - Provided that the matching **MPI_Send** actually sent fewer elements
- **MPI_Get_count** can be used to know how many elements of type *datatype* have actually been received
- See mpi-get-count.c

Blocking communication and deadlocks

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



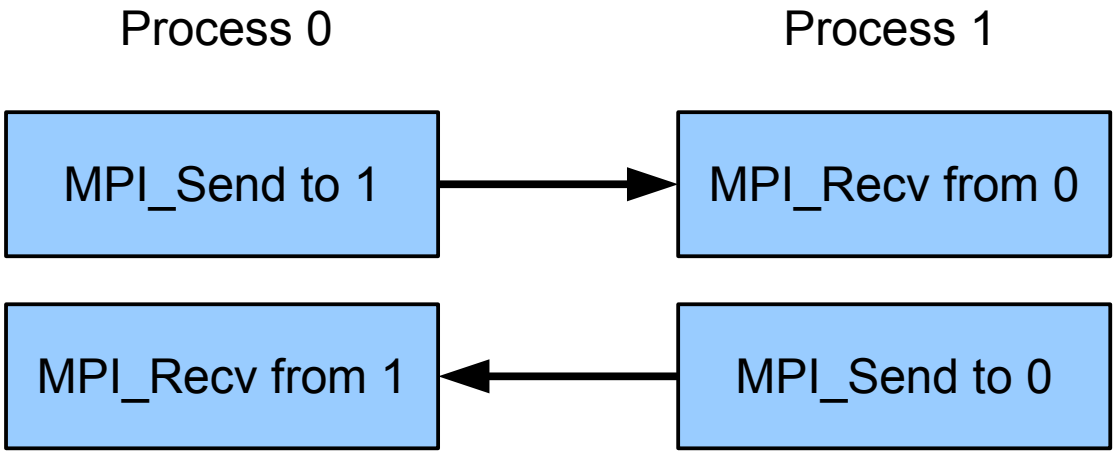
Blocking communication and deadlocks



….

Blocking communication and deadlocks

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



- ...or use non-blocking communication primitives
- ...or use **`MPI_Sendrecv()`** if appropriate (see later)

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 cal **`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

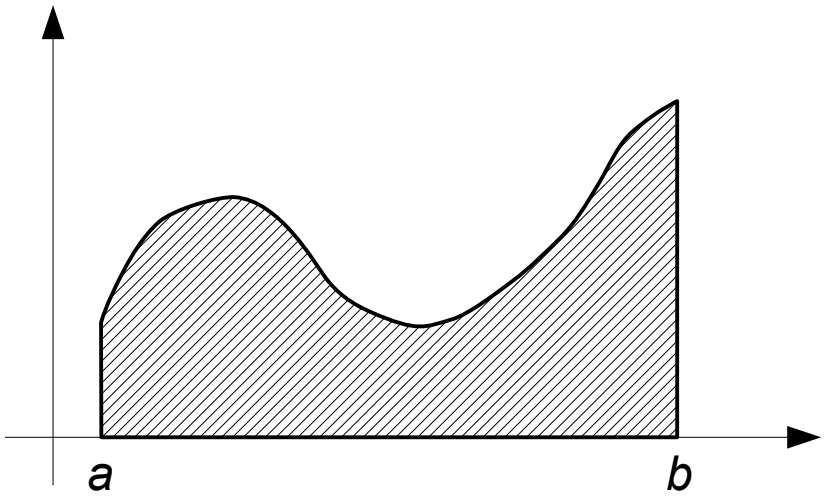
```
/* mpi-async.c */
#include <stdio.h>
#include <mpi.h>
int main( int argc, char *argv[])
{
    int rank, size, buf;
    MPI_Status status;
    MPI_Request req;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    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;
}
```

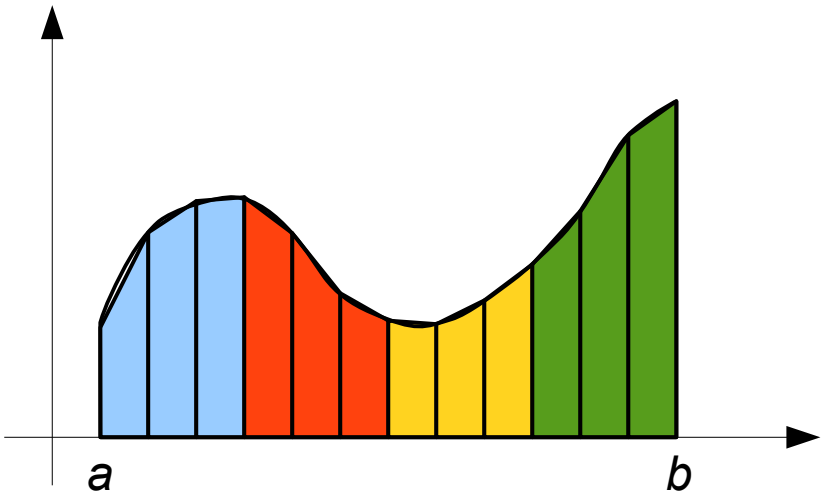
Aborting the computation

- 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)

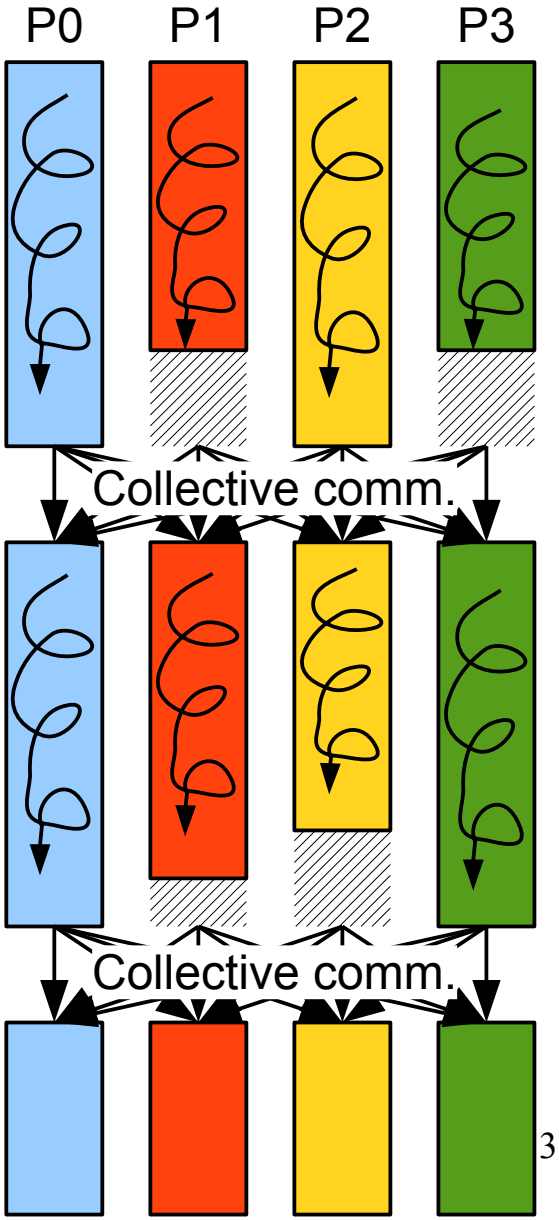
```
partial_result = trap(my_rank, comm_sz, a, b, n);

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



Credits: Tim Mattson

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

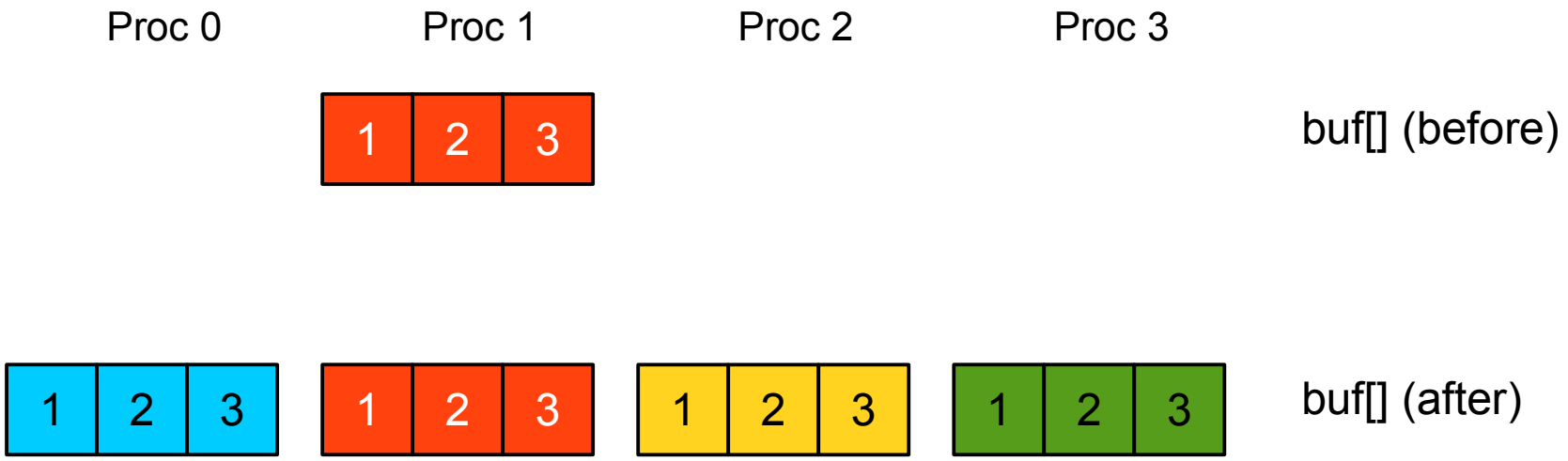
MPI_Barrier()

- 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

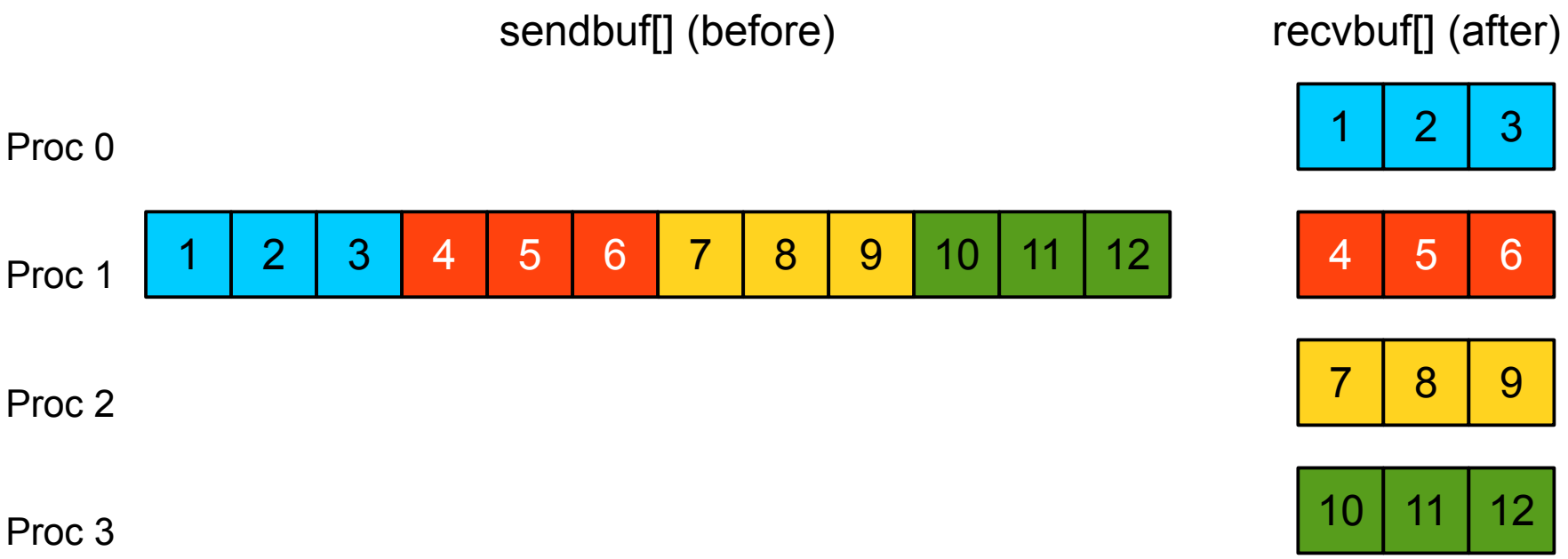
```
count = 3;
src = 1; /* broadcast originates from process 1 */
MPI_Bcast(buf, count, MPI_INT, src, MPI_COMM_WORLD);
```



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);
```



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, ...)
```

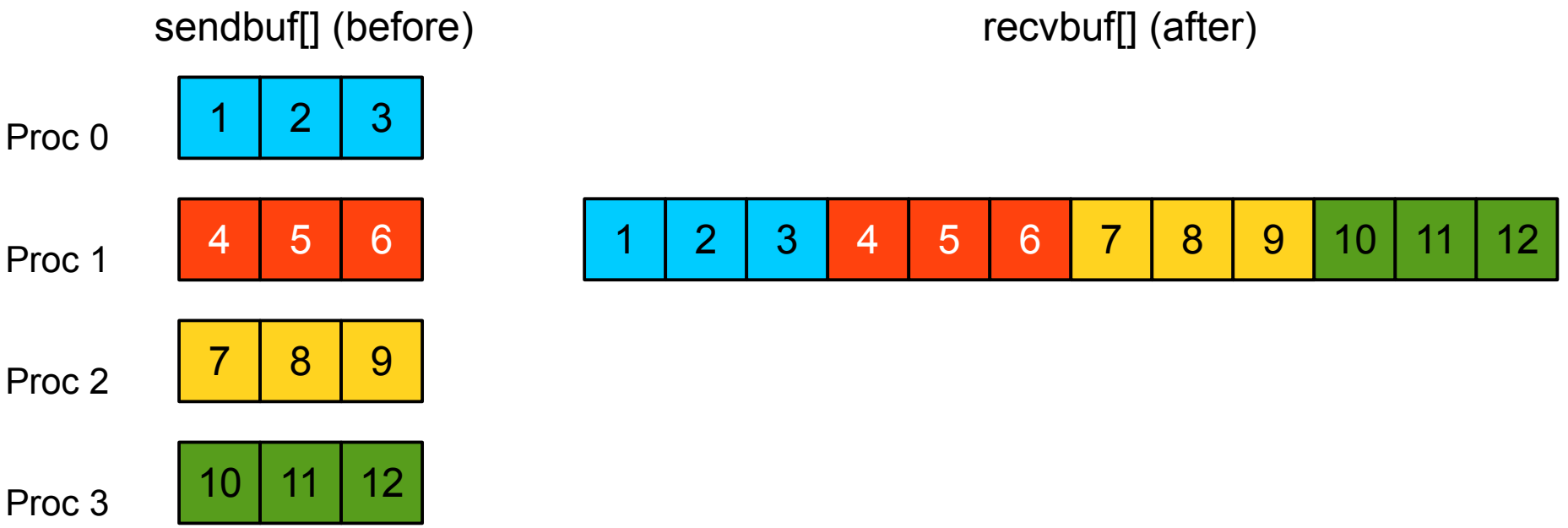
and all other processes execute

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

MPI_Gather()

Gathers together data from other processes

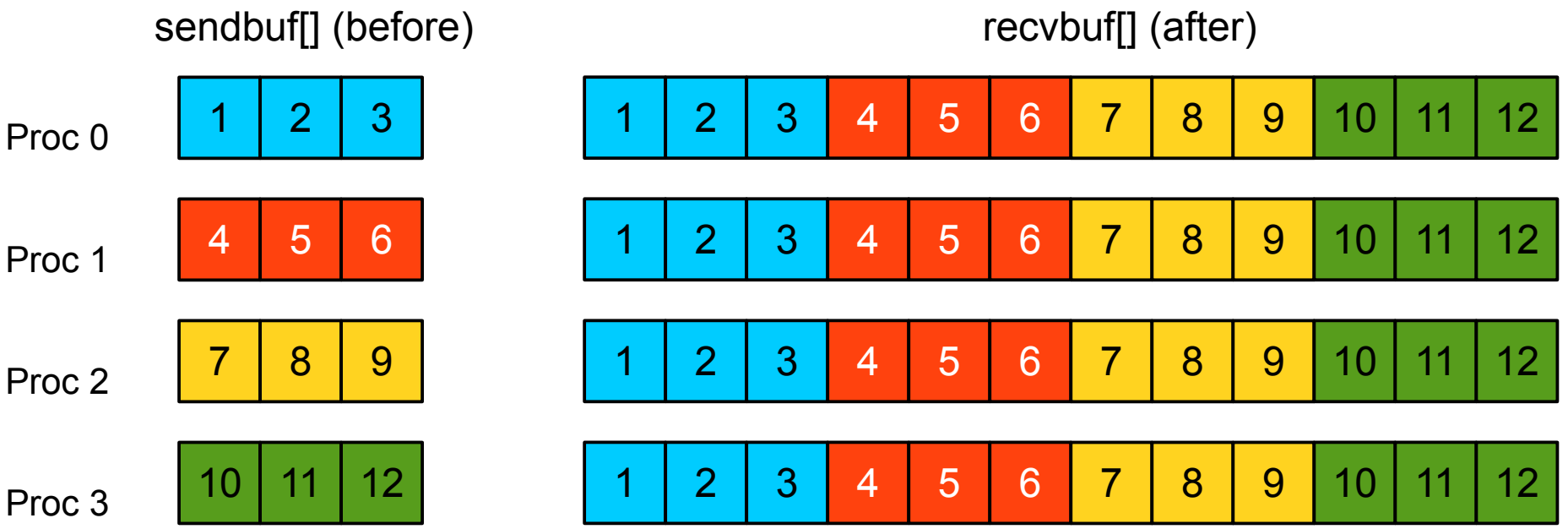
```
sendcnt = 3; /* how many items are sent by each process */
recvcnt = 3; /* how many items are received from each process */
dst = 1;     /* message will be gathered at process 1 */
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT, dst, MPI_COMM_WORLD);
```



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);
```

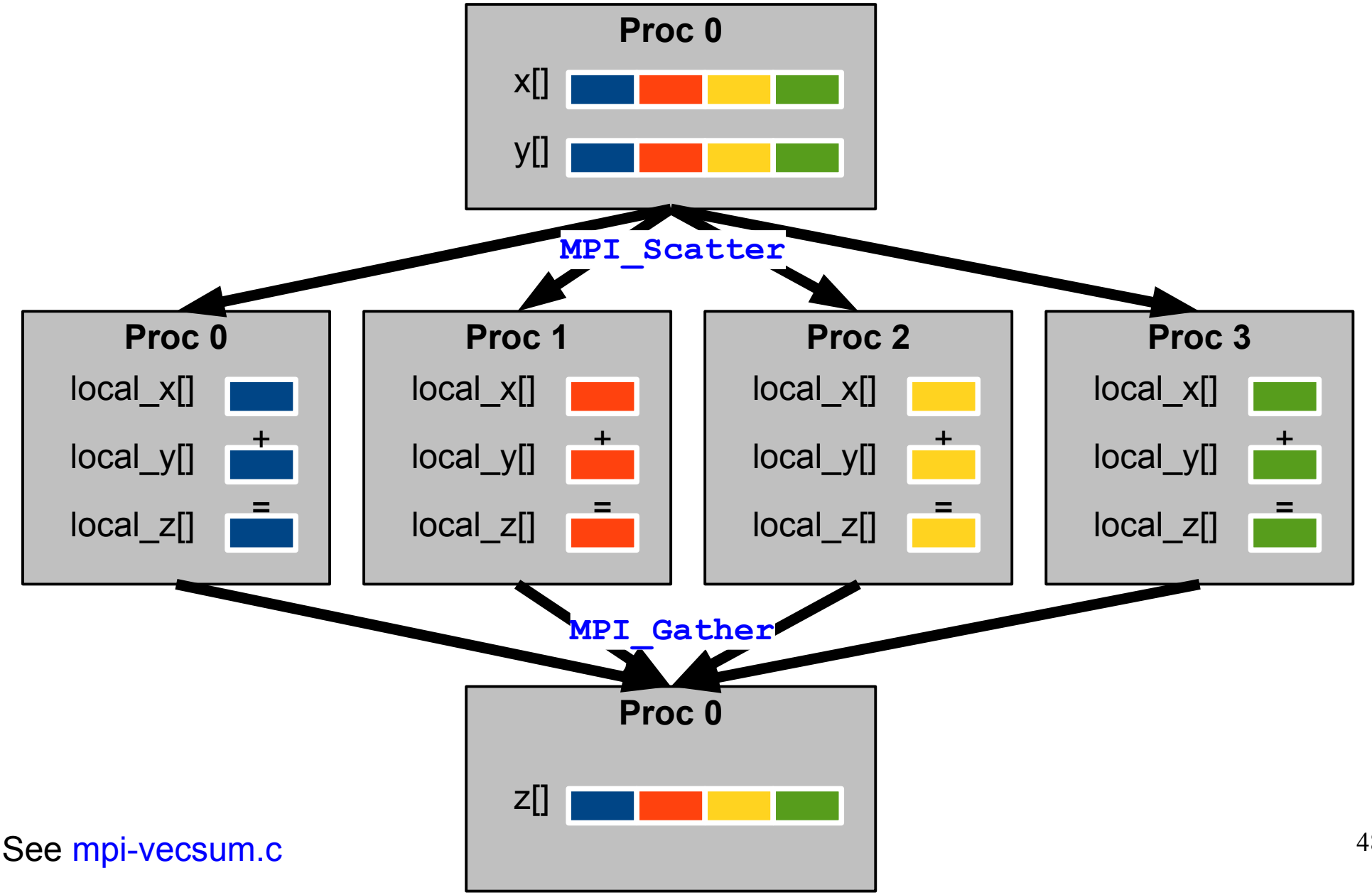


Example: Vector Sum

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

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

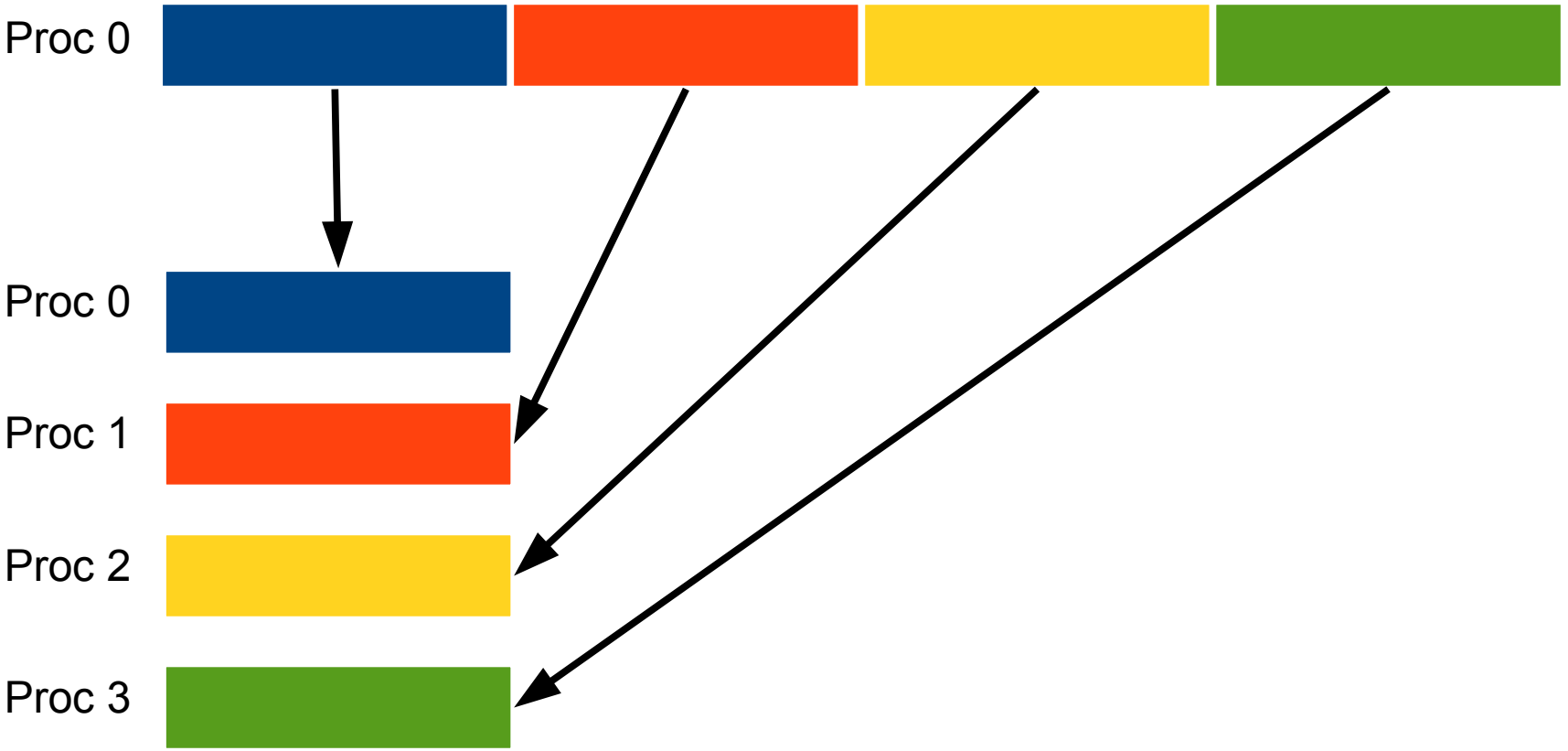
Parallel Vector Sum



See mpi-vecsum.c

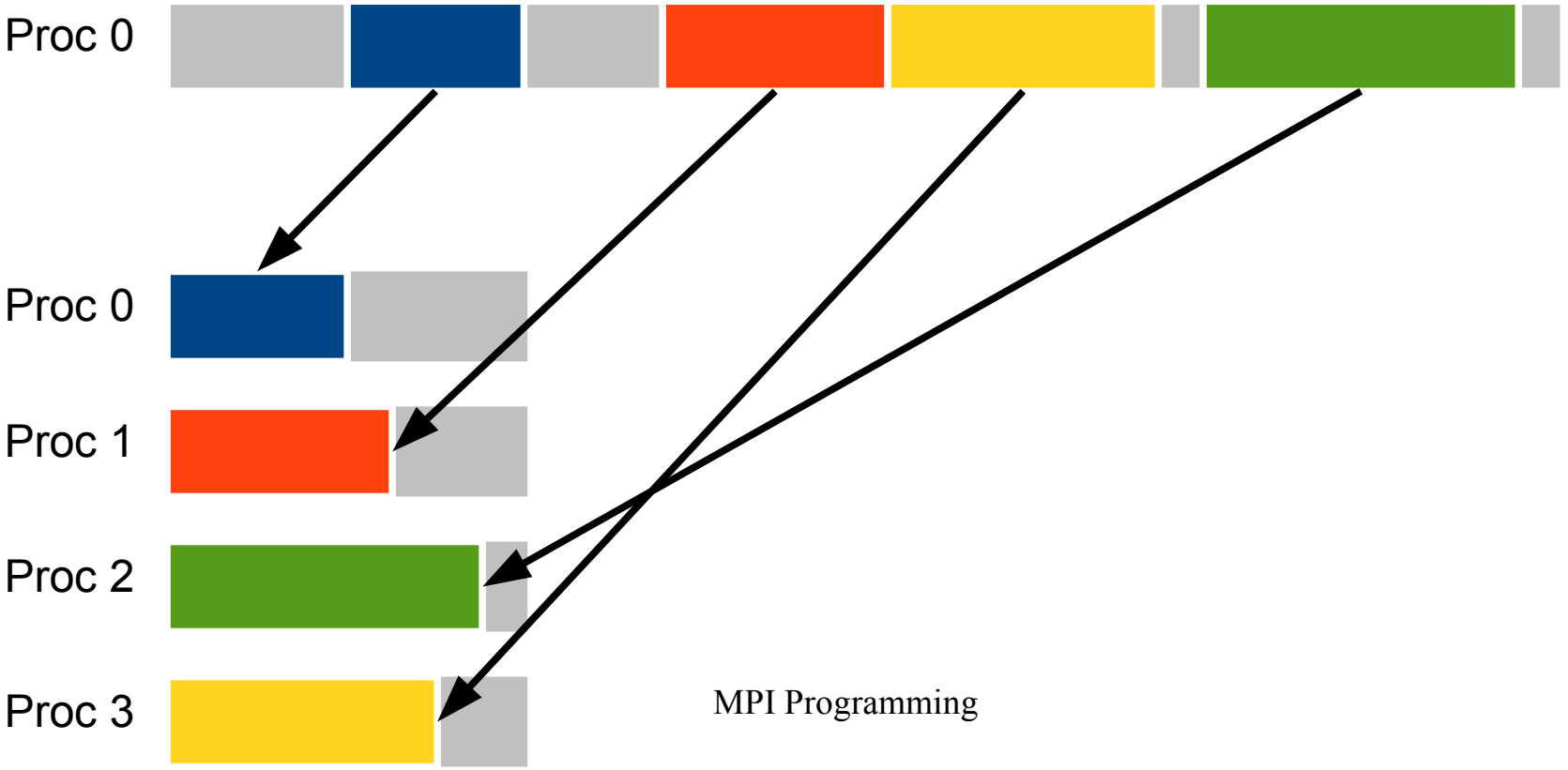
MPI_Scatter()

- Contiguous data
- Uniform message size

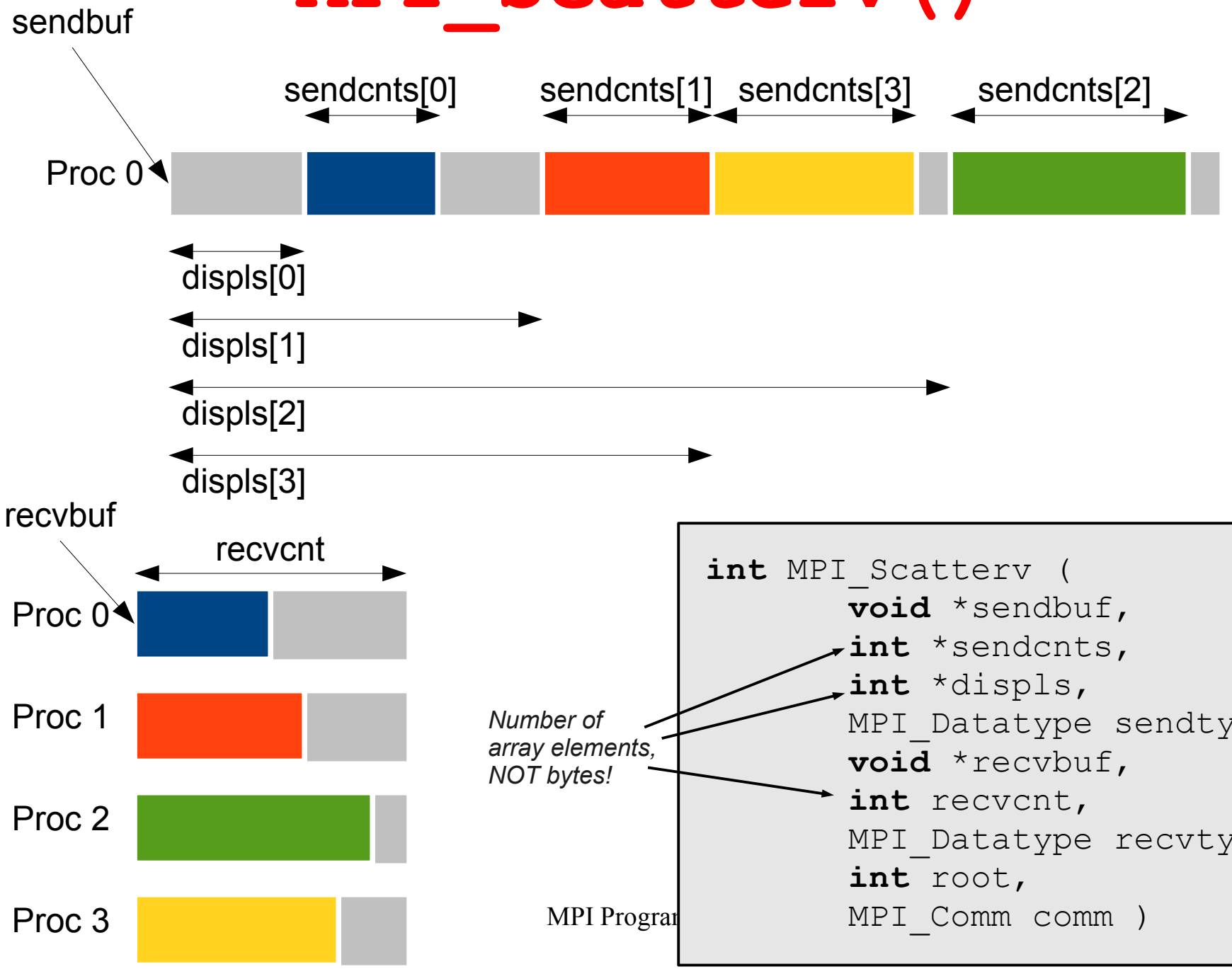


MPI_Scatterv() / MPI_Gatherv()

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



MPI_Scatterv()

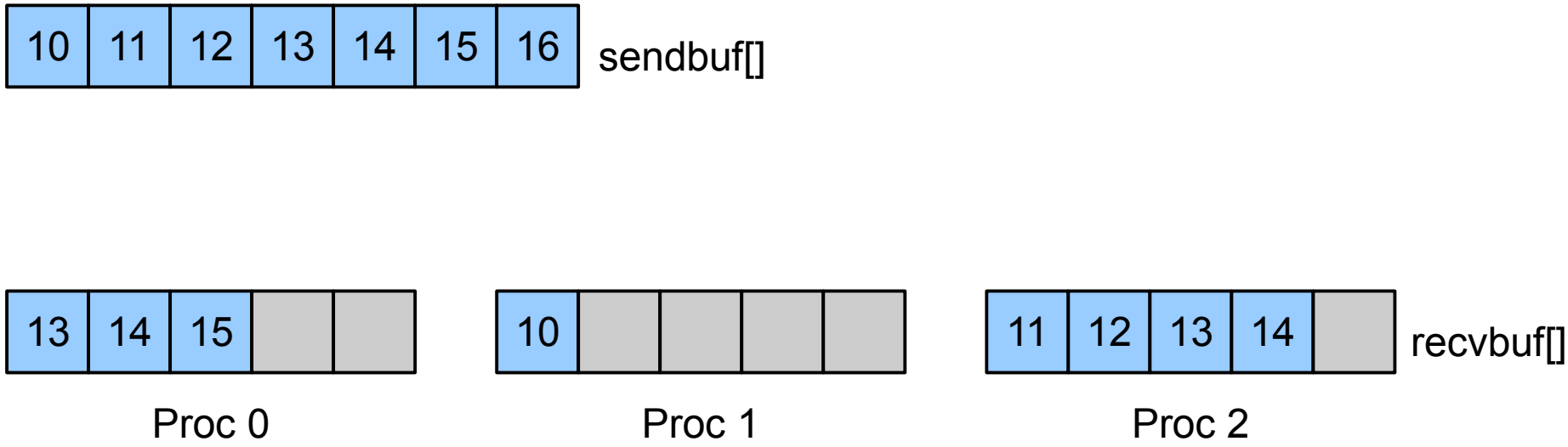


Number of array elements, NOT bytes!

```
int MPI_Scatterv (
        void *sendbuf,
        int *sendcnts,
        int *displs,
        MPI_Datatype sendtype,
        void *recvbuf,
        int recvcnt,
        MPI_Datatype recvtype,
        int root,
        MPI_Comm comm )
```

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);
```

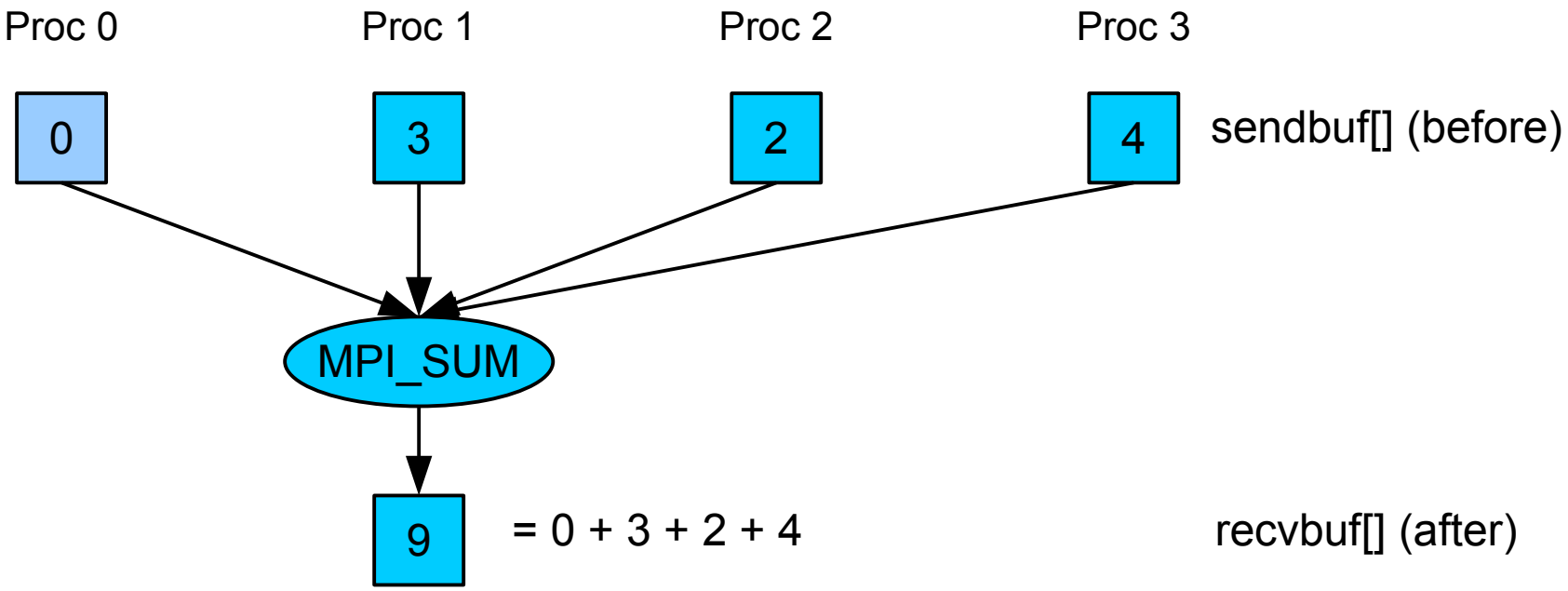


See mpi-vecsum3.c

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);
```



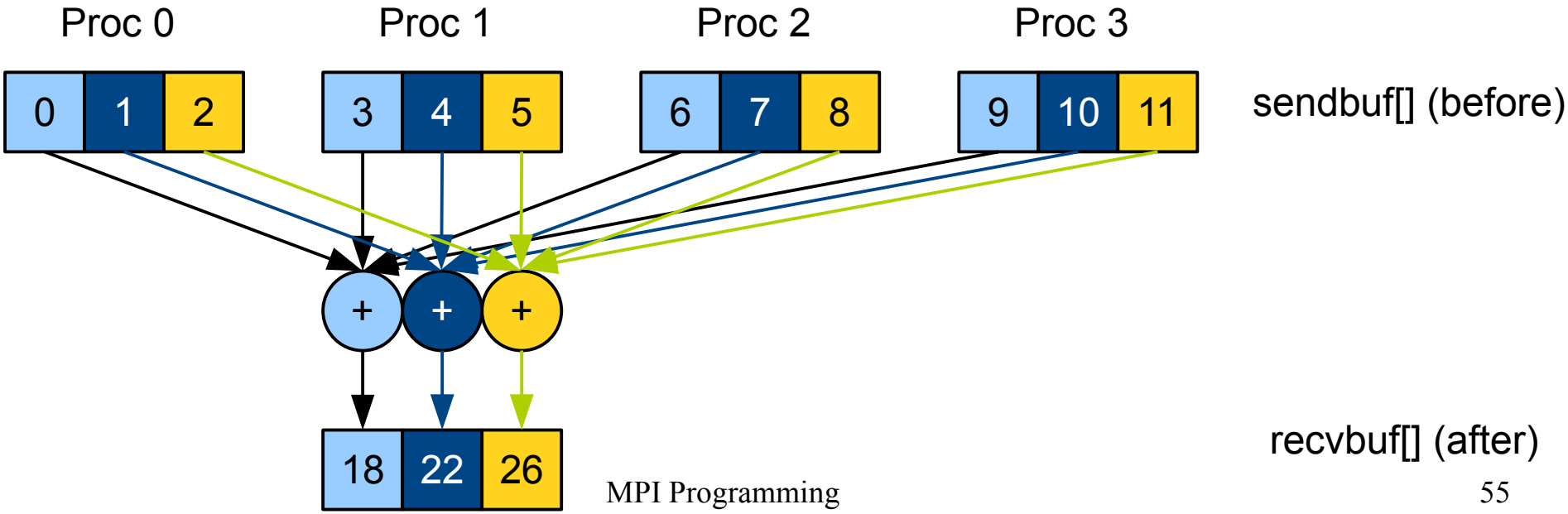
Predefined reduction operators

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

MPI_Reduce()

- If 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);
```



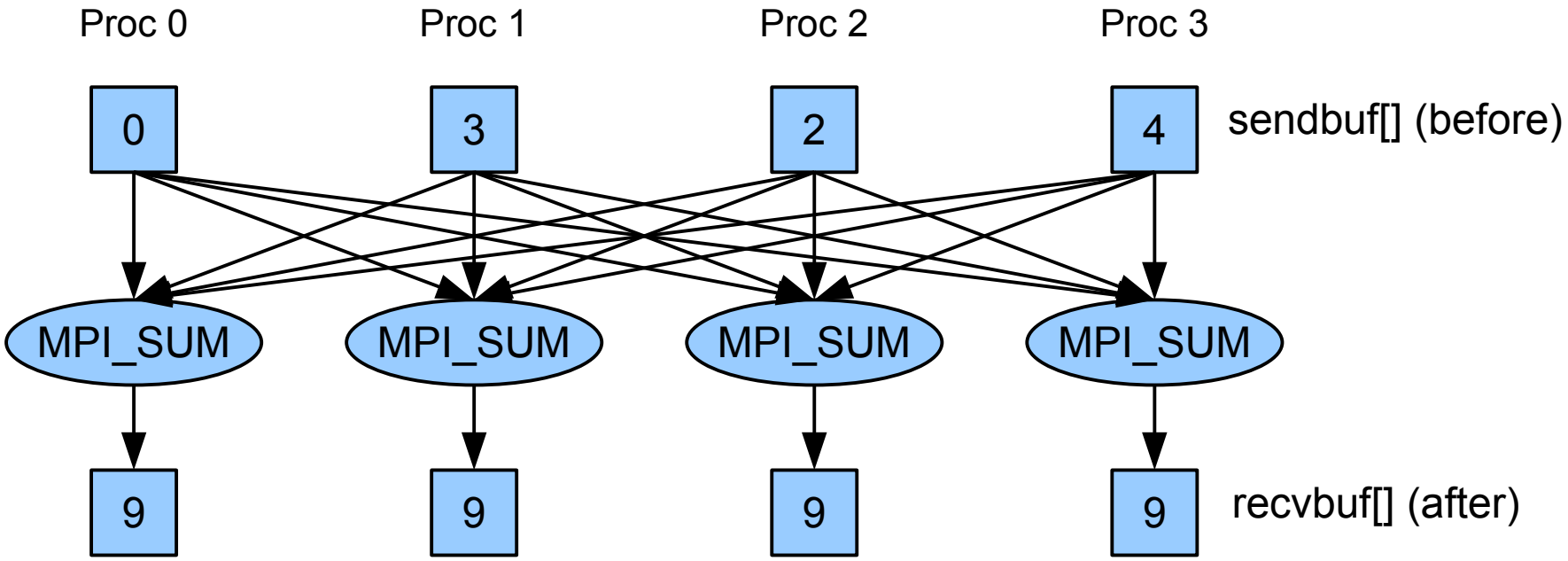
Parallel trapezoid (with reduction)

- See mpi-trap1.c

MPI_Allreduce()

Performs a reduction and place result in all processes

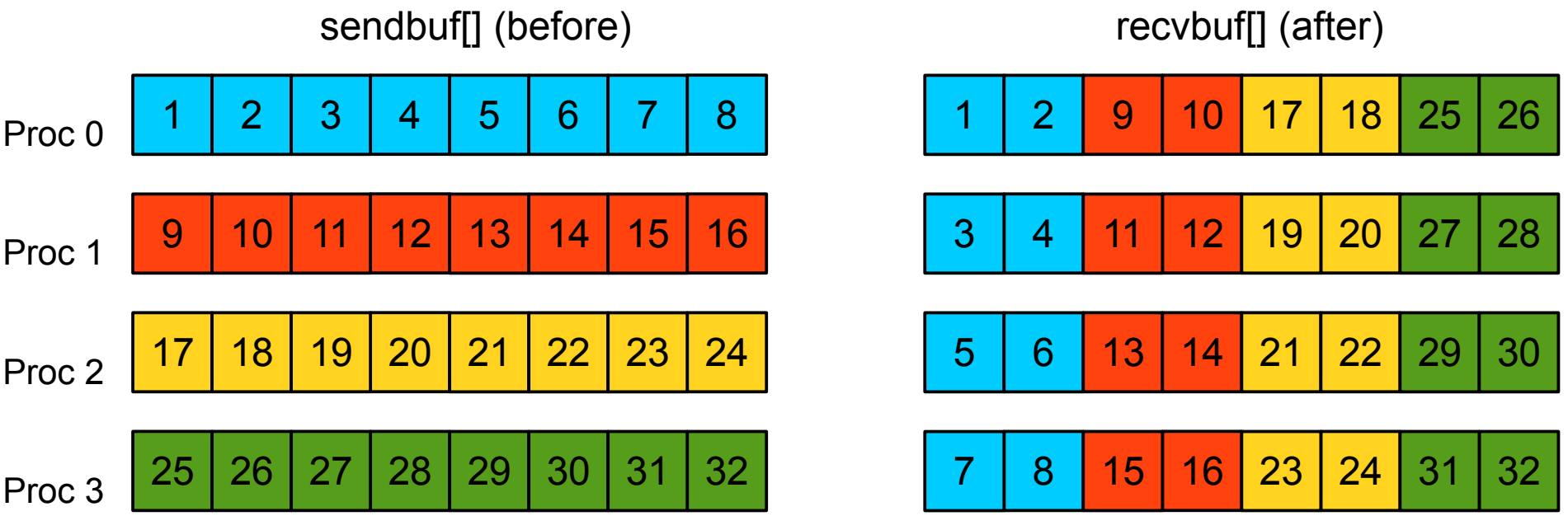
```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



MPI_Alltoall()

Each process performs a scatter operation

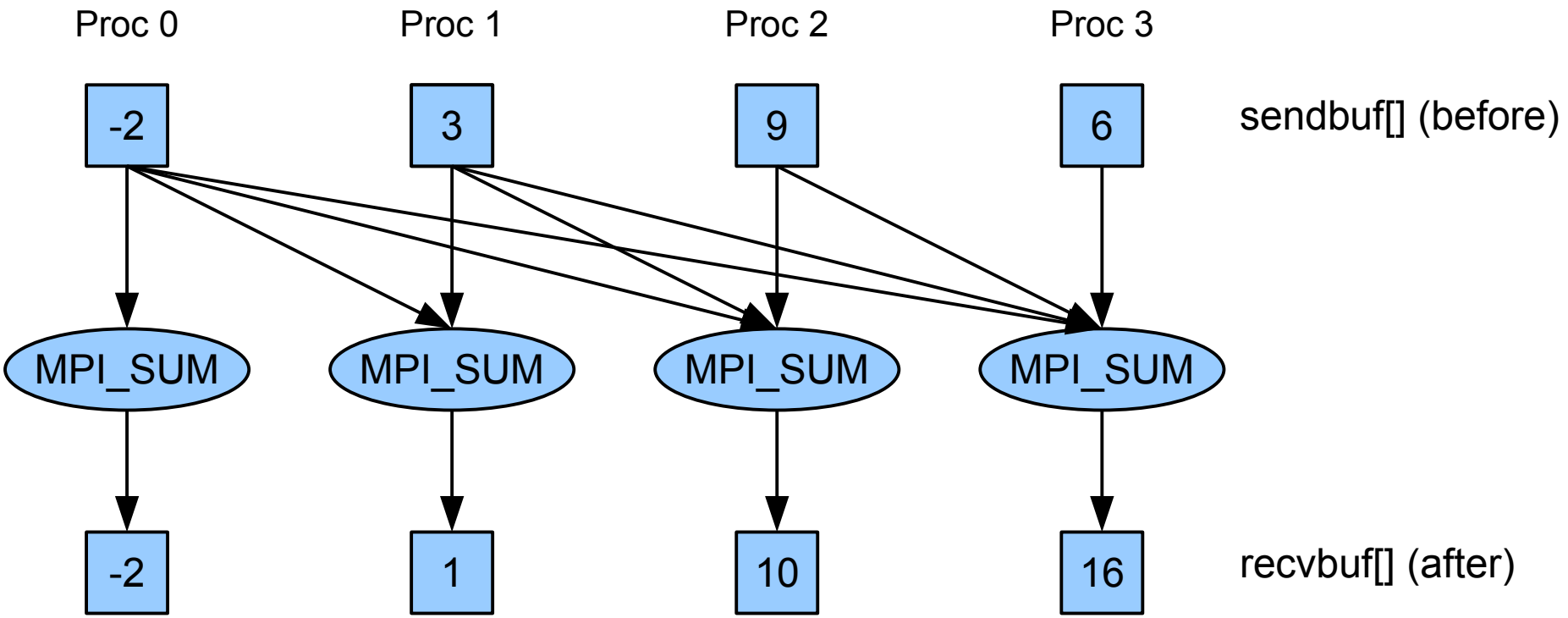
```
sendcnt = 2;
recvcnt = 2;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT, MPI_COMM_WORLD);
```



MPI_Scan()

Inclusive scan

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

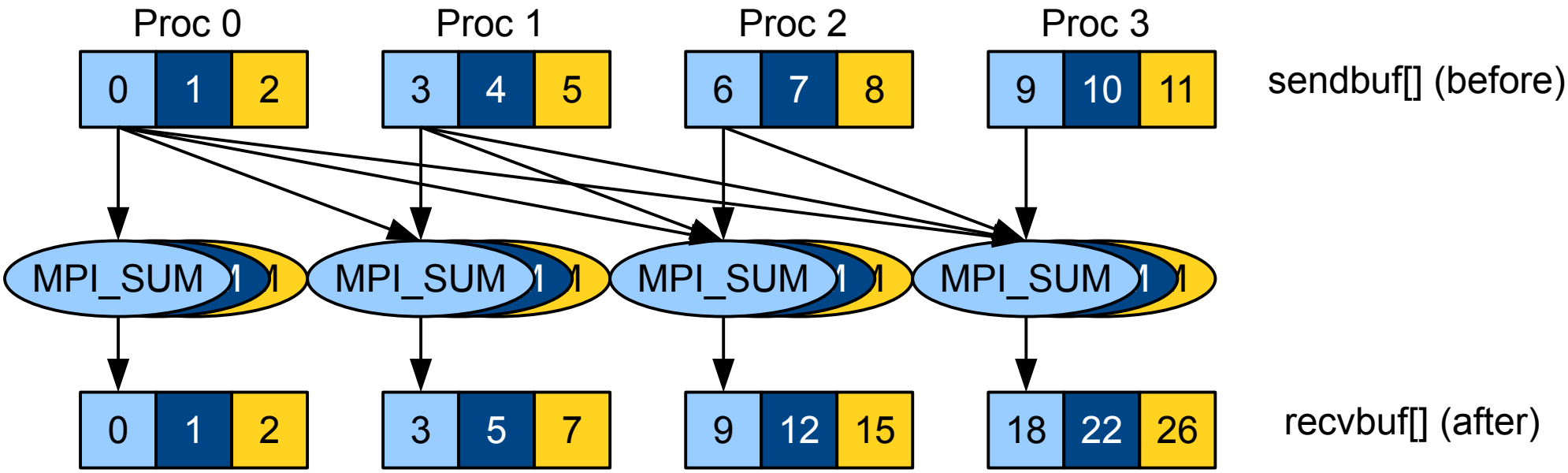


See mpi-scan.c

MPI_Scan()

- If 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);
```



See mpi-scan.c

Collective Communication Routines / 1

MPI_Barrier(comm)

- Synchronization operation. Creates a barrier synchronization in a group. Each rocess, 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 logicaly 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.

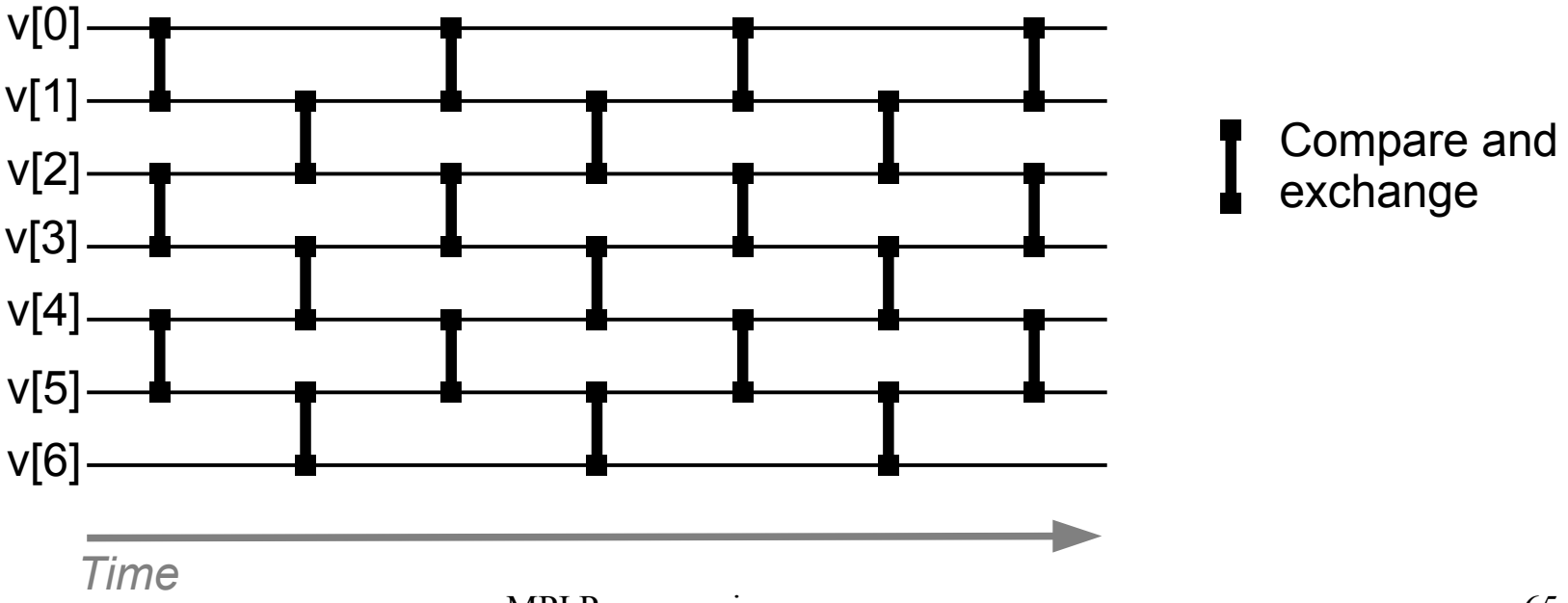
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm)

- 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



Choosing the Granularity

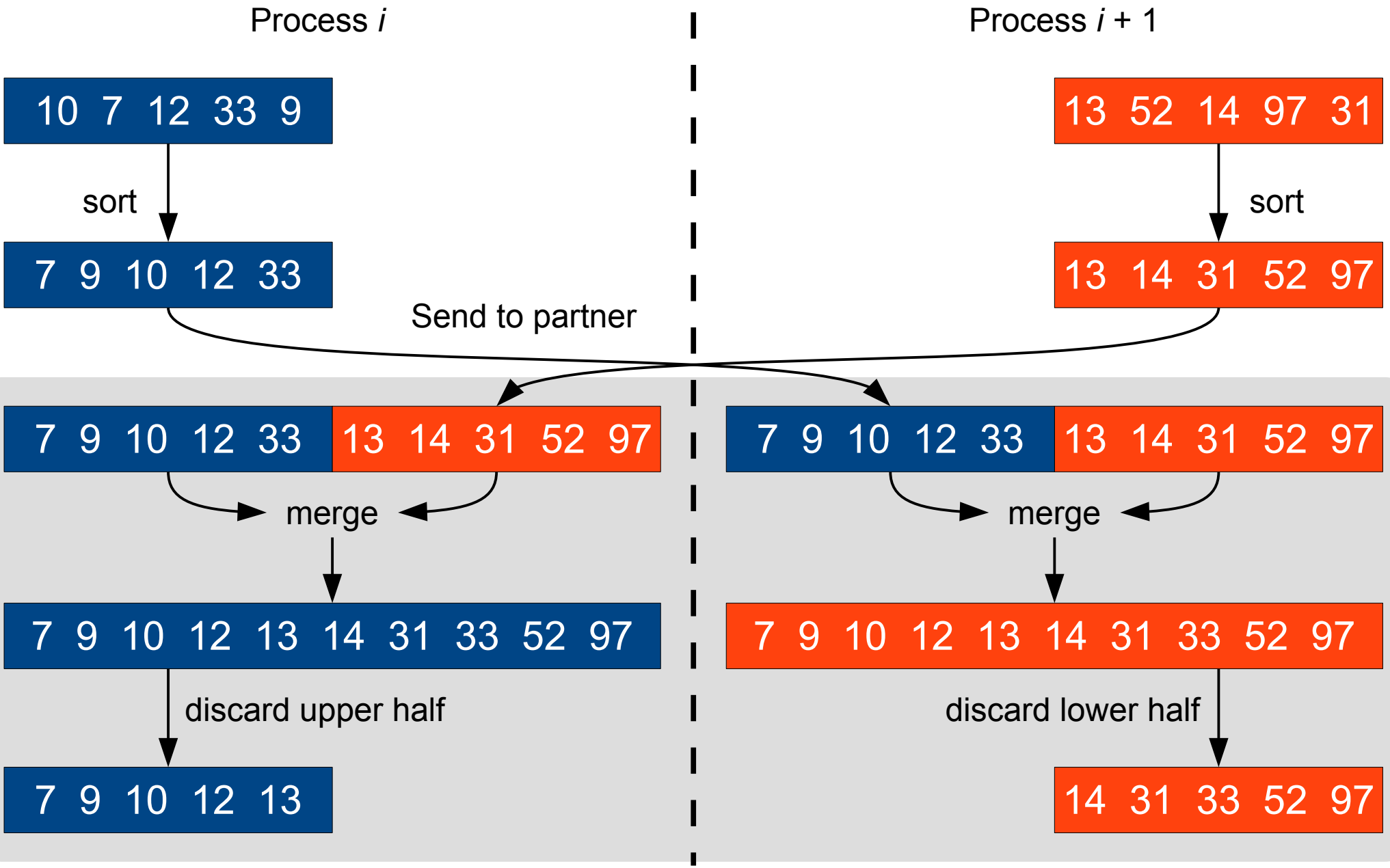
- Communication in distributed memory systems is very expensive, therefore working on individual array elements is not appropriate
- We achieve a coarser granularity by splitting the array into blocks and working at the block level

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

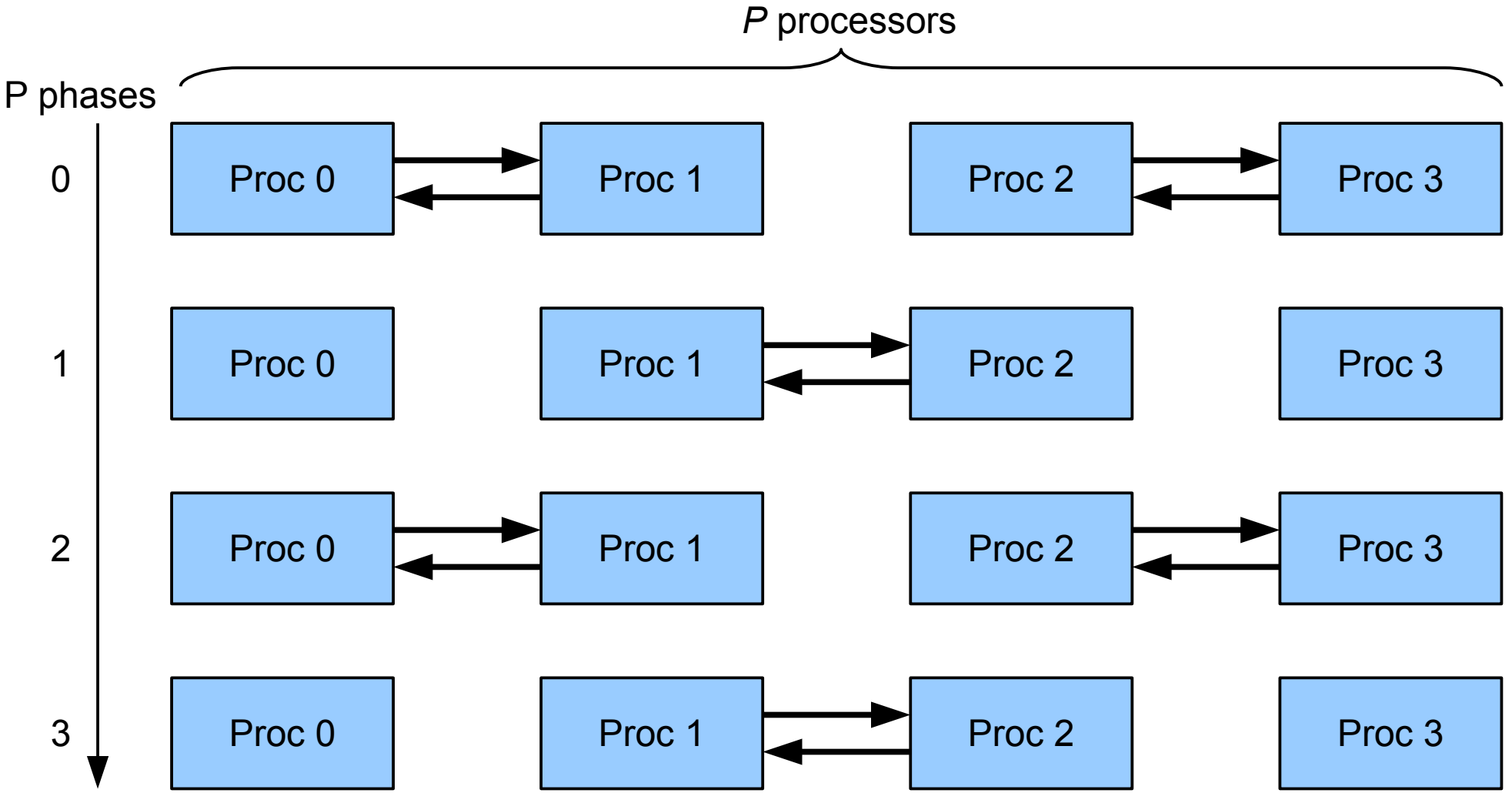
- The array is split in blocks that are assigned to MPI processes
- Each process sorts its block (e.g., using **`qsort`**)
- At each exchange-swap step:
  - Process *i* sends a copy of its chunk to process *i* + 1
  - Process *i* + 1 sends a copy of its chunk to process *i*
  - Process *i* merges the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

Odd-Even Transposition Sort

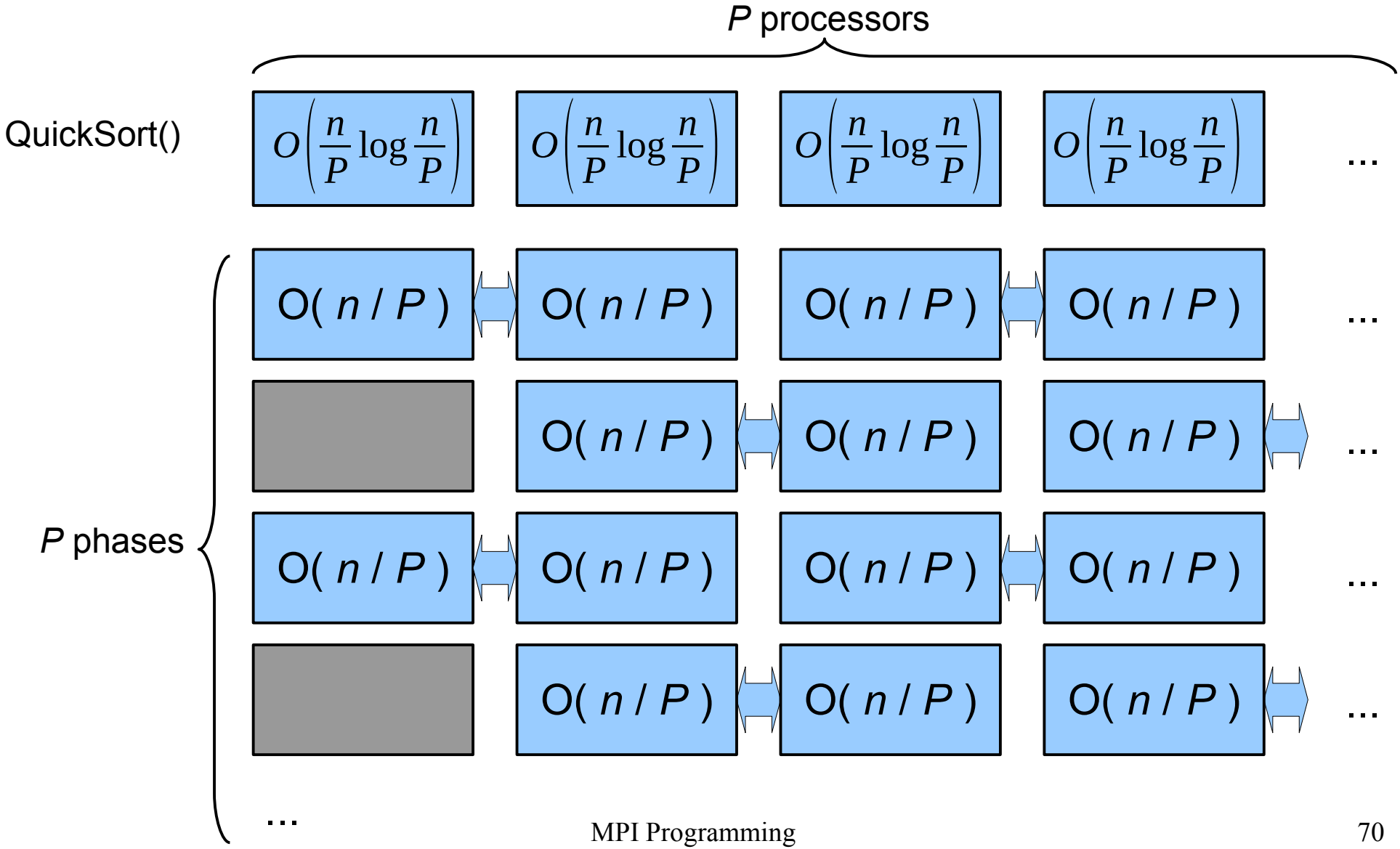


....

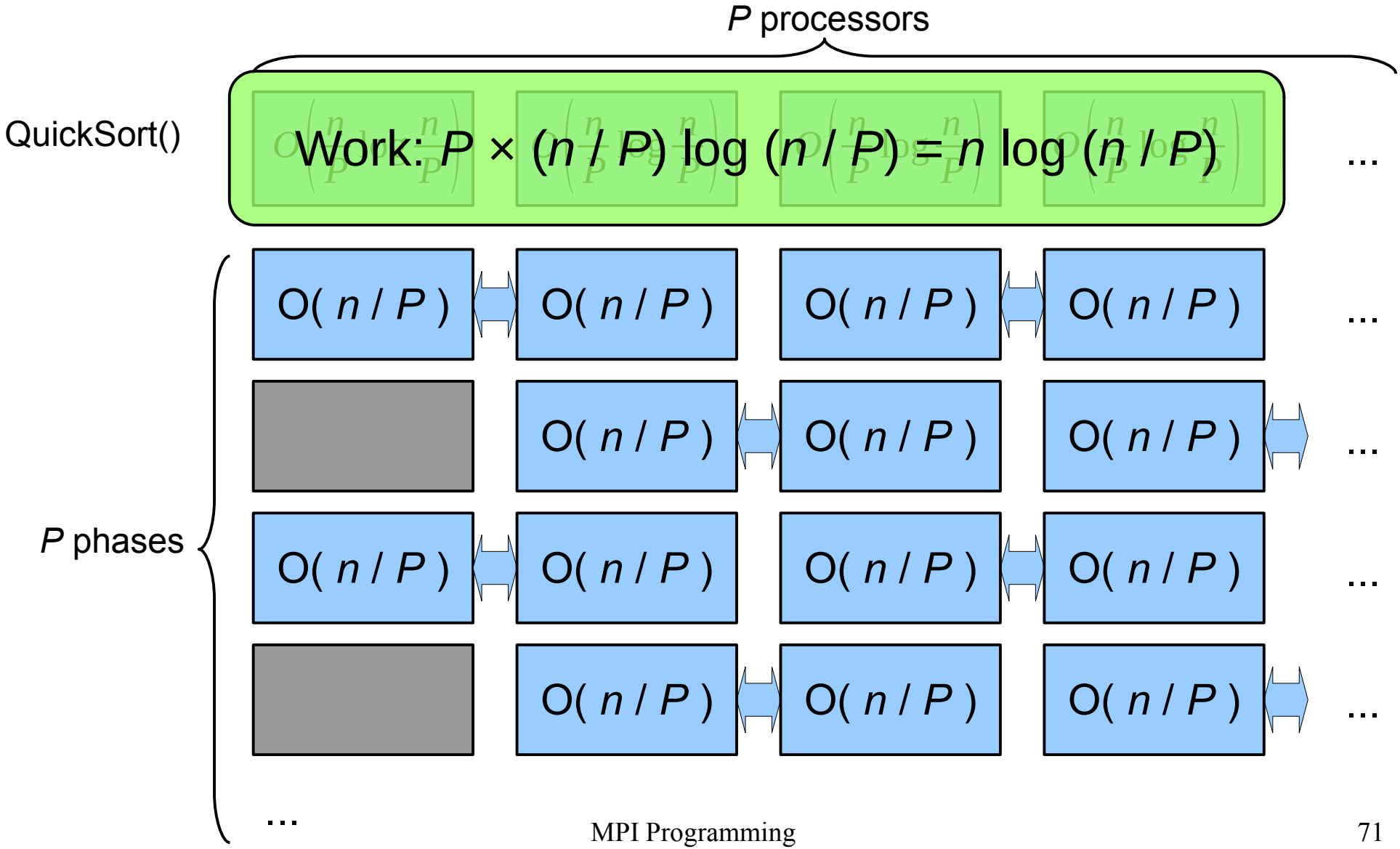
Communication pattern for odd-even sort



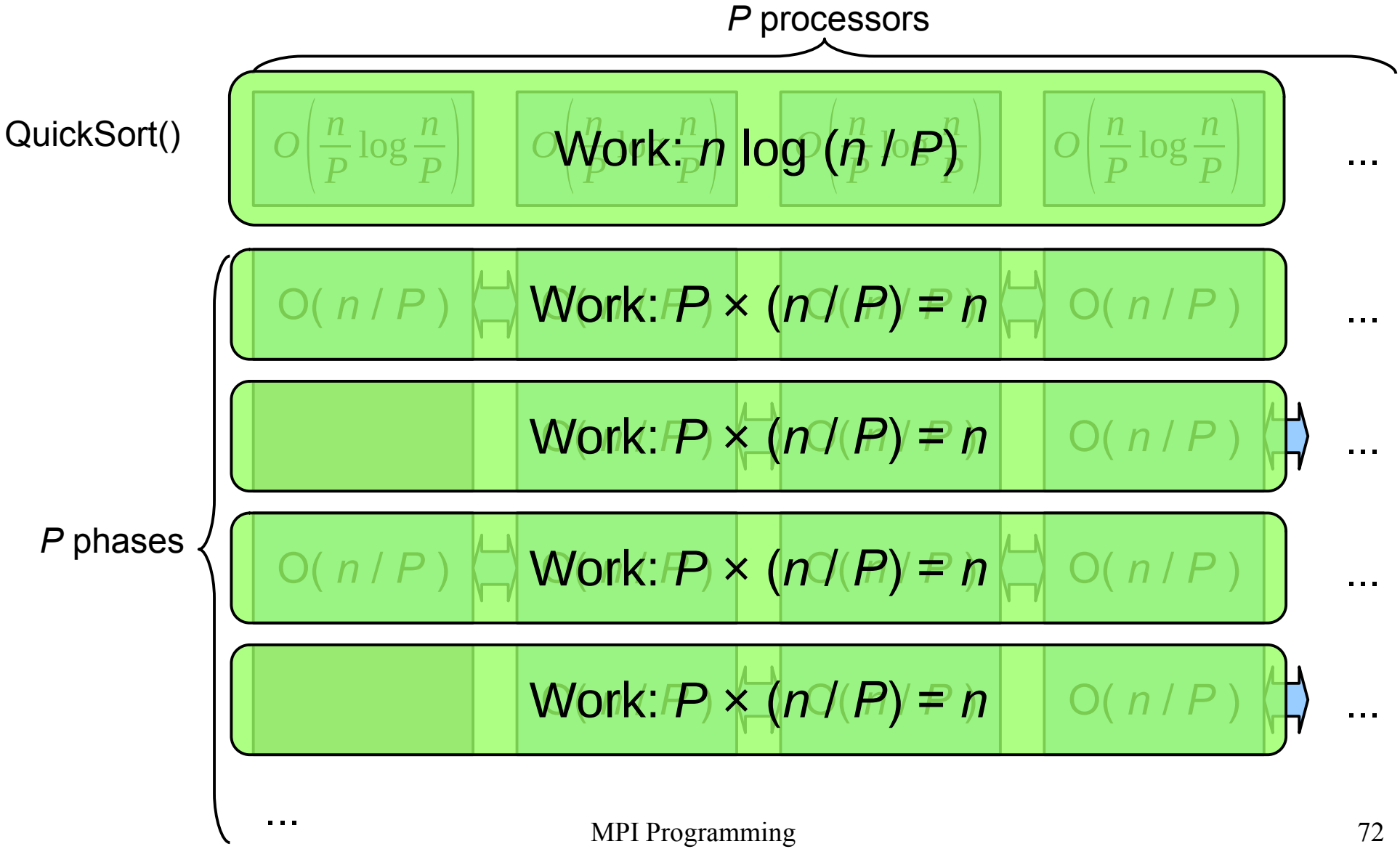
Odd-Even Transposition Sort: Analysis (Work)



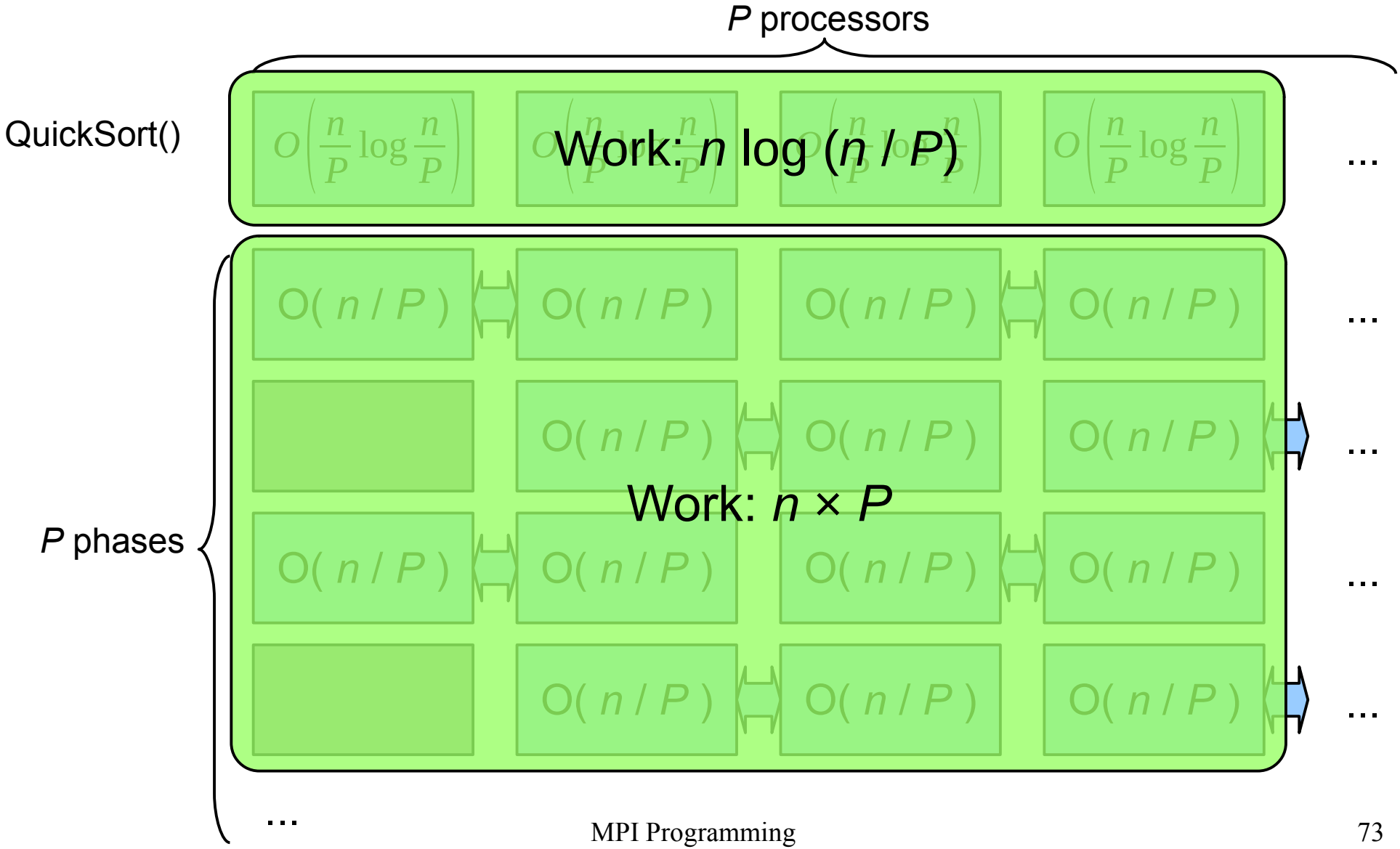
Odd-Even Transposition Sort: Analysis (Work)



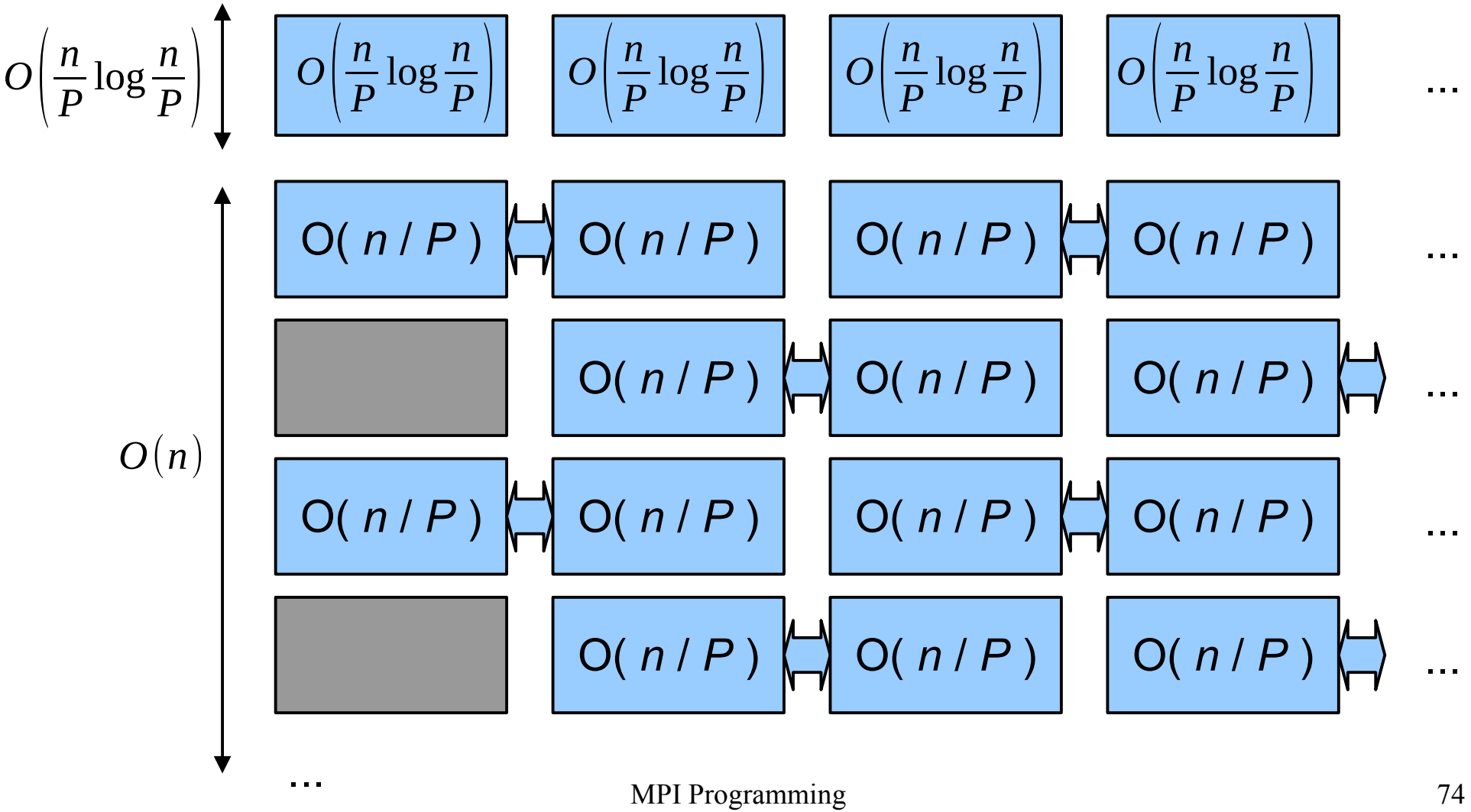
Odd-Even Transposition Sort: Analysis (Work)



Odd-Even Transposition Sort: Analysis (Work)



Odd-Even Transposition Sort: Analysis (Time)



Odd-Even Transposition Sort: Analysis

- Total time: O((n / P) log (n / P) + n)
  - If we use P = 1 processor
    - Time: $n \log n$
  - If we use $P = n$ processors
    - Time: n
- Total work: O(n log (n / P) + $n P$)
  - If we use P = 1 processor
    - Work: $n \log n$
  - If we use $P = n$ processors
    - Work: n^2
- If we increase P
  - The time decreases
  - The work increases

Odd-Even Transposition Sort: Analysis

- This algorithm can not be directly compared with the OpenMP version, since here we have a phase that uses QuickSort
- How much can we increase *P* to keep the work wtill proportional to *n* log *n*?
  - We need to solve $n \log (n / P) + n P = n \log n$ for the unknown *P*

Odd-Even Transposition Sort: Analysis

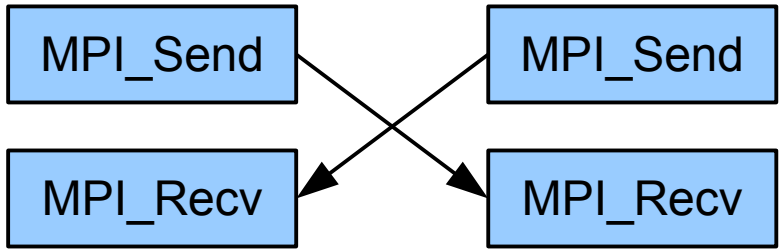
- $n \log (n / P) + n P = n \log n$

  Let us try $P = \log n$
  - $n \log (n / \log n) + n \log n =? n \log n$
  - $n \log n – n \log (\log n) + n \log n =? n \log n$ Yes
- We can increase the number of processors up to $\log n$ and still get optimal work
- Using $P = \log n$ processors the time will be $O((n / \log n) (\log n – \log \log n) + n) \sim O(n)$

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);
}
```

Solution 2 (better): **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

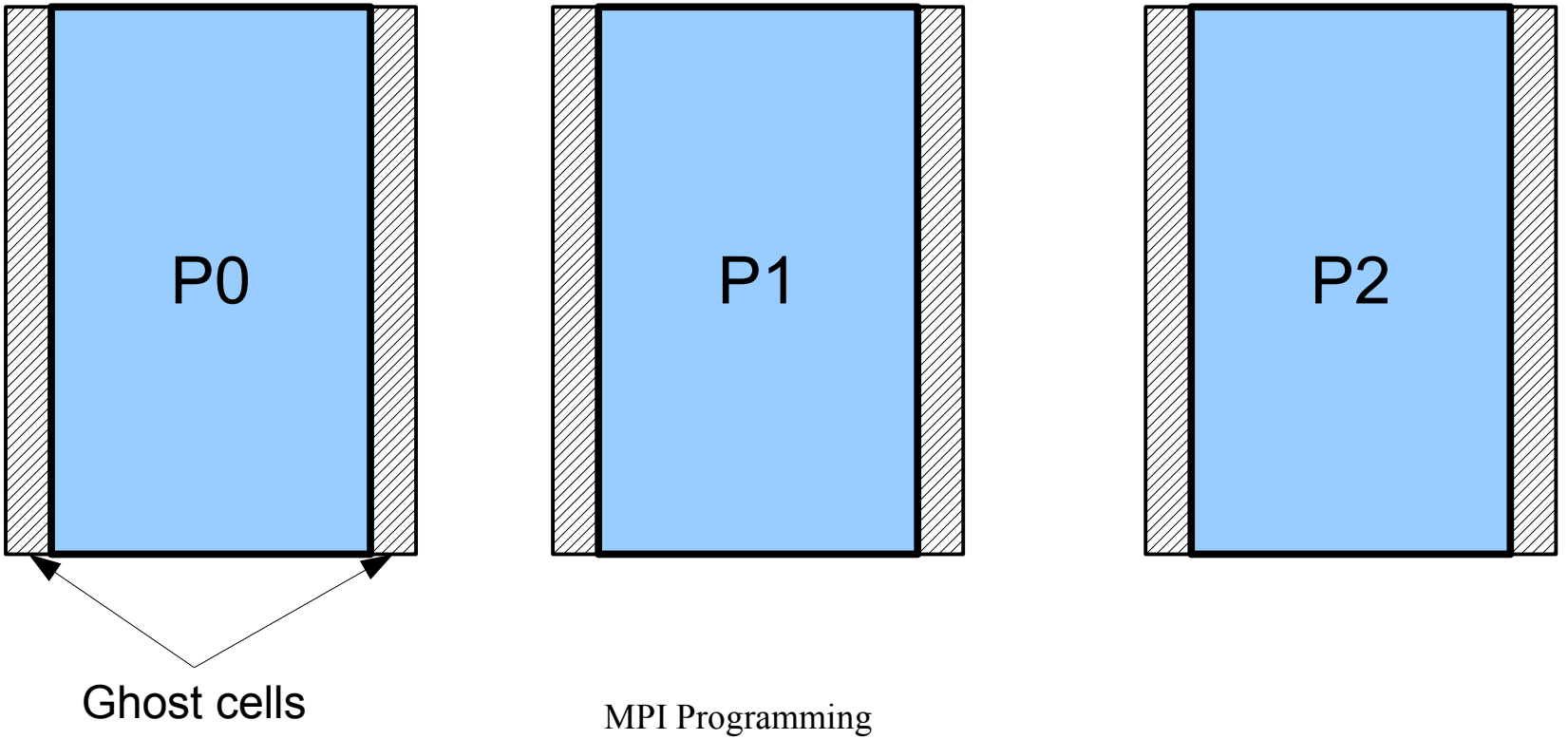
```
int MPI_Sendrecv(
    void*          sendbuf,
    int            sendcount,
    MPI_Datatype   sendtype,
    int            dest,
    int            sendtag,
    void*          recvbuf,
    int            recvcount,
    MPI_Datatype   recvtype,
    int            source,
    int            recvtag,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

See mpi-odd-even.c

MPI Datatypes

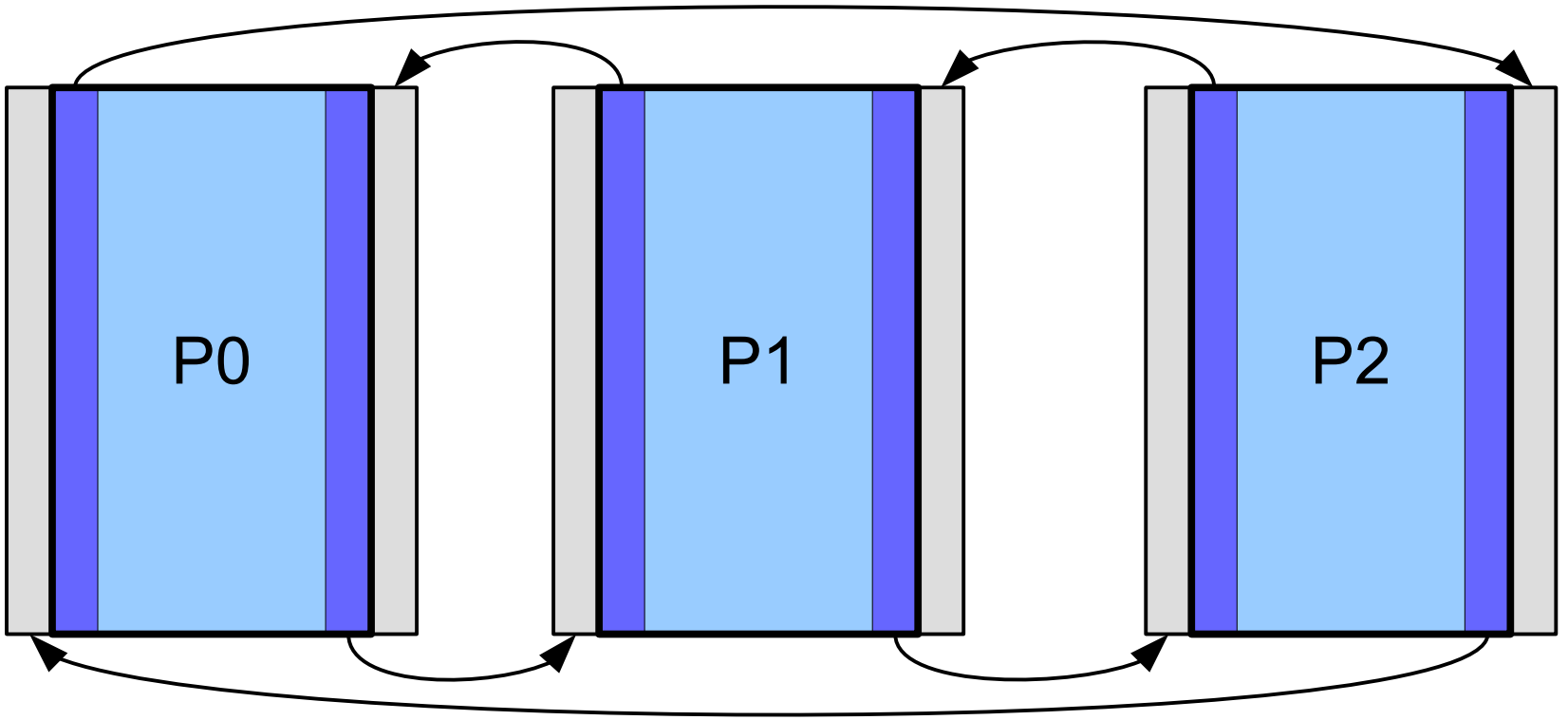
Example

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



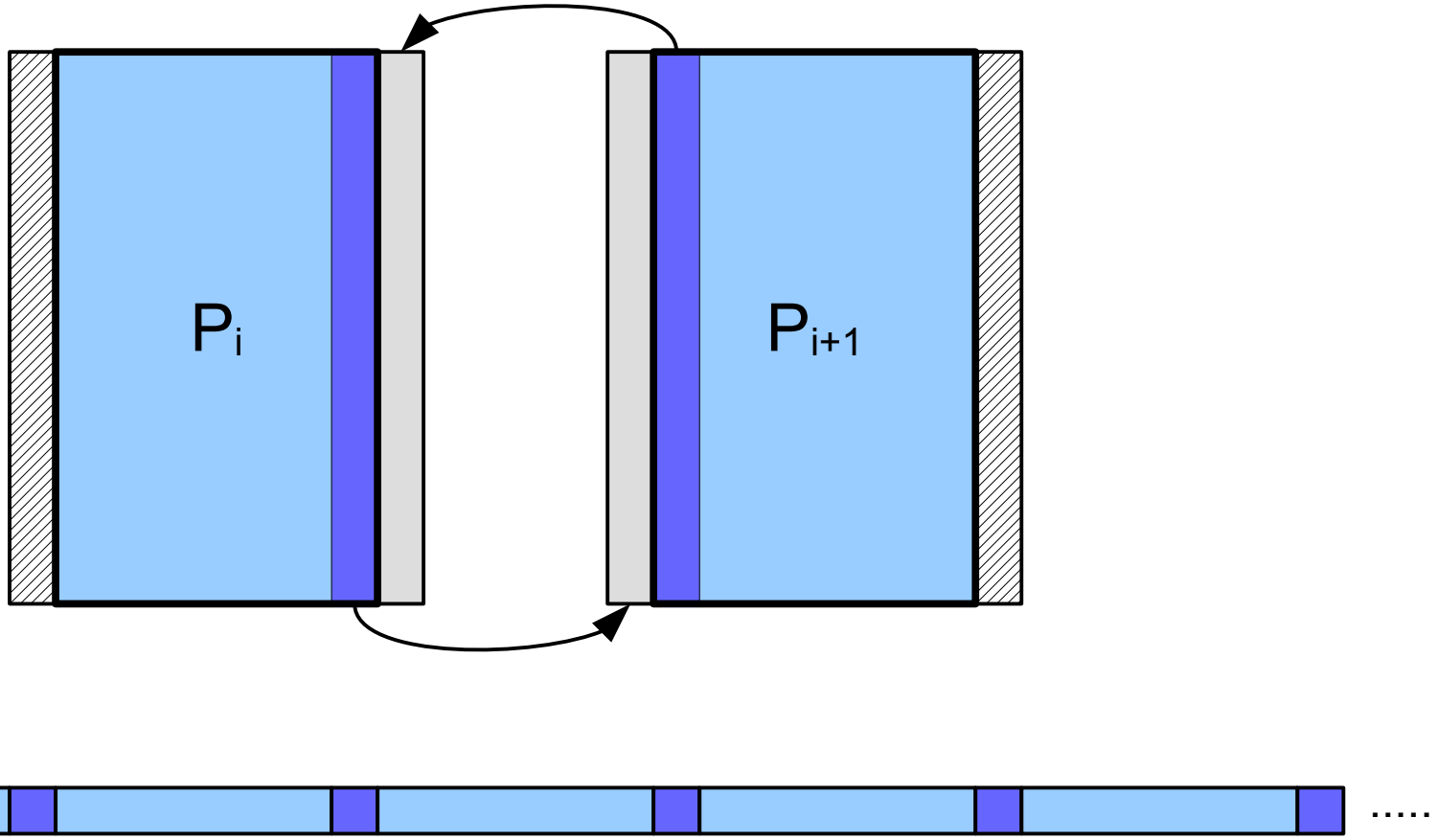
Example

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



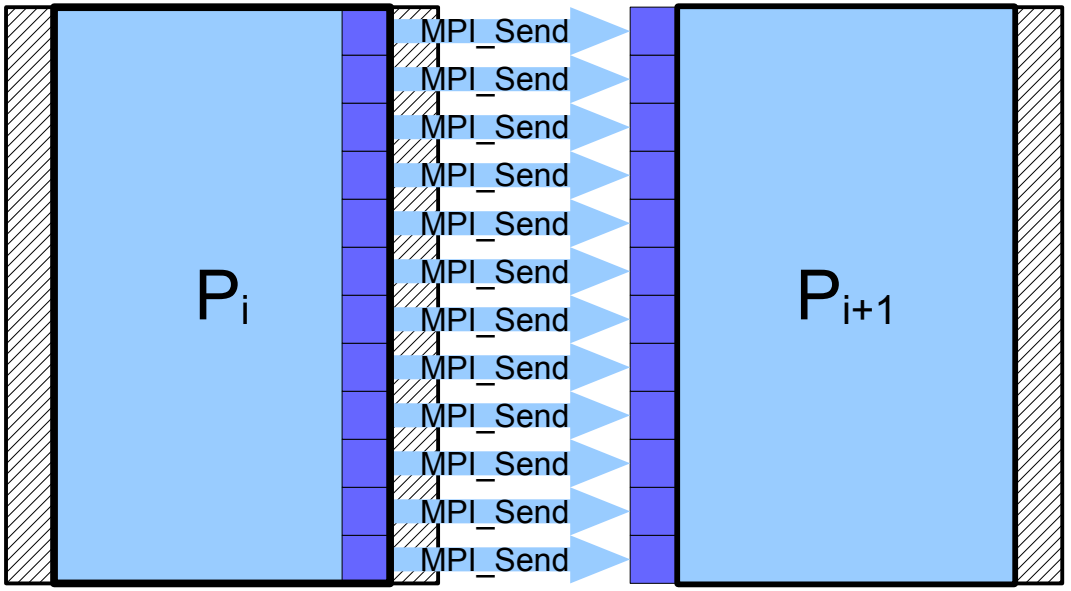
Example

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



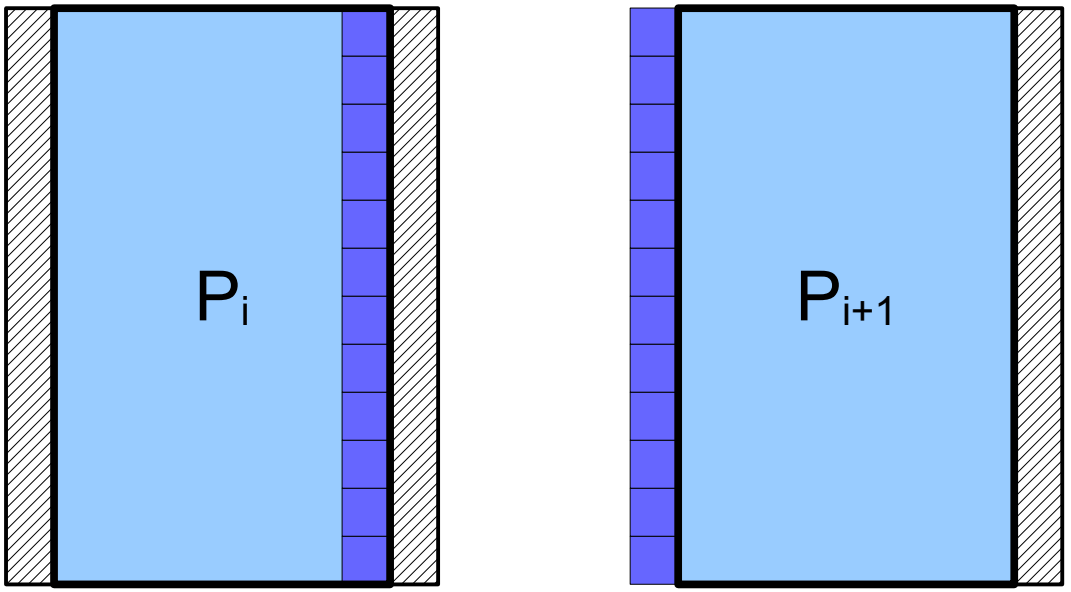
Example

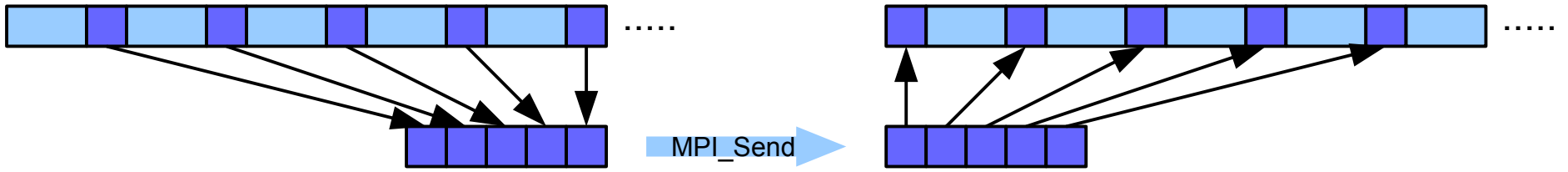
- **The BAD** solution: send each element with **`MPI_Send`** (or **`MPI_Isend`**)



Example

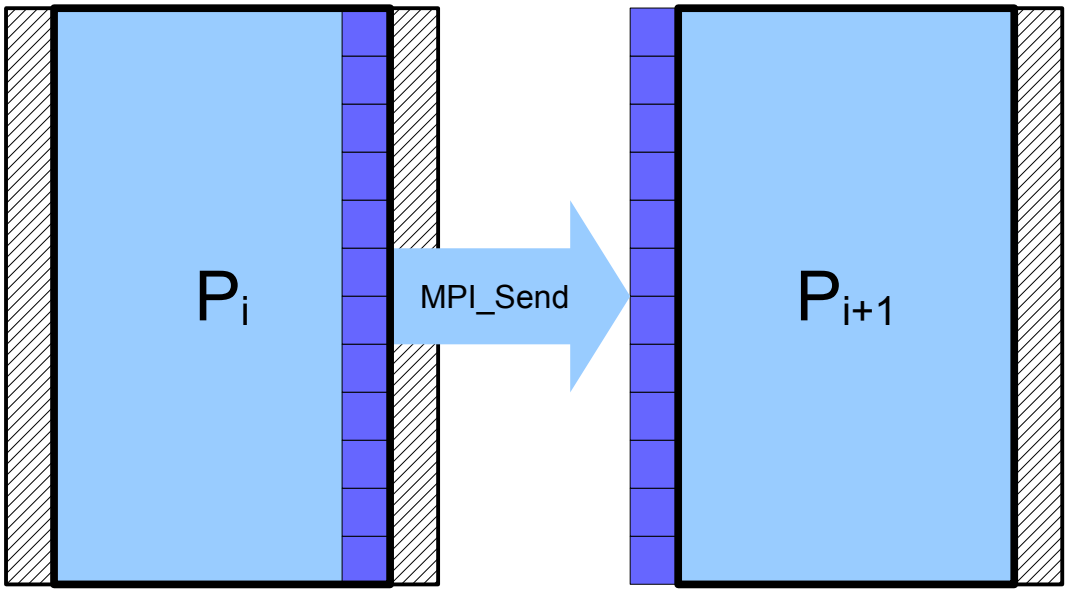
- **The UGLY** solution: copy the column into a temporary buffer; **`MPI_Send()`** the buffer; fill the destination column





Example

- **The GOOD** solution: define a new datatype for the column, and **`MPI_Send`** the column directly



MPI Derived Datatypes

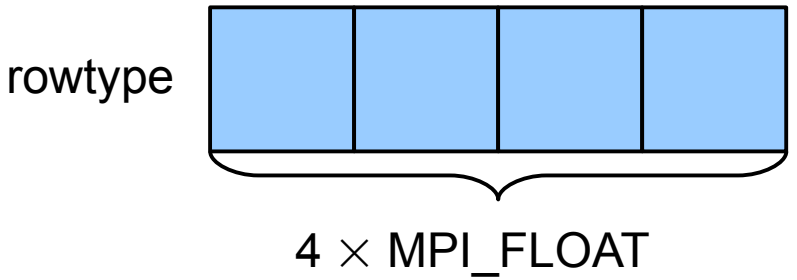
- MPI provides several functions for constructing derived data types:
  - Contiguous: a contiguous block of elements
  - Vector: a strided vector of elements
  - Indexed: an irregularly spaced set of blocks of the same type
  - Struct: an irregularly spaced set of blocks possibly of different types
- 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)

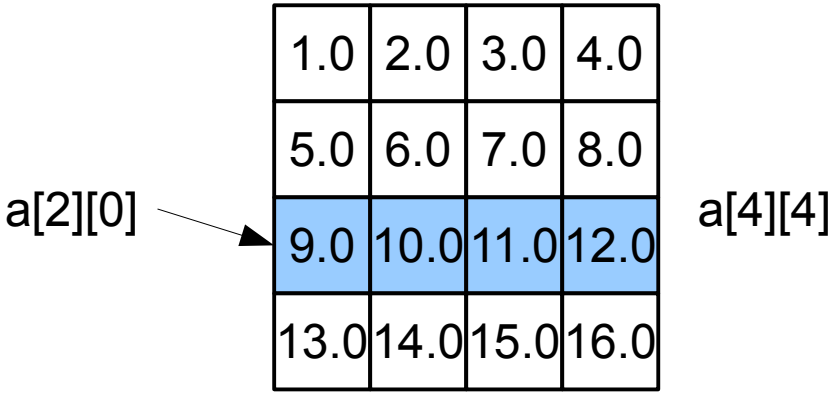
- A contiguous block of count elements of an existing type oldtype
  - oldtype can be another previously defined custom datatype



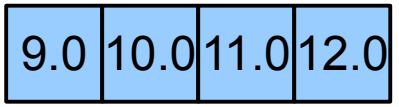
MPI_Type_contiguous()

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

See mpi-type-contiguous.c




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



1 element of type rowtype

MPI_Type_vector()

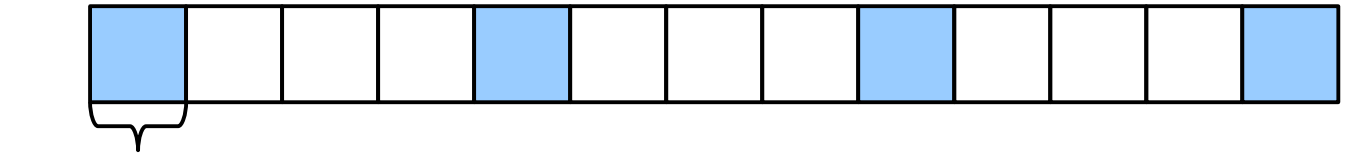
```
MPI_Datatype columntype;
MPI_Type_vector( 4, 1, 4, MPI_FLOAT, &columntype );
MPI_Type_commit(&columntype);
```

```
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 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);
```



blocklen = 1

stride = 4

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, columntype, dest, tag, MPI_COMM_WORLD);
```

2.0	6.0	10.0	14.0

1 element of type columntype

See mpi-type-vector.c

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

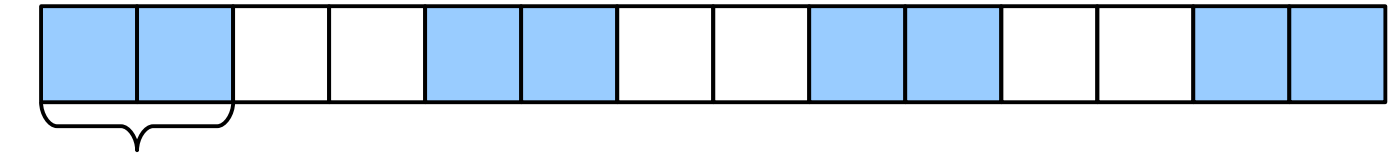
a[4][4]

```
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);
```



blocklen = 2

stride = 4

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	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

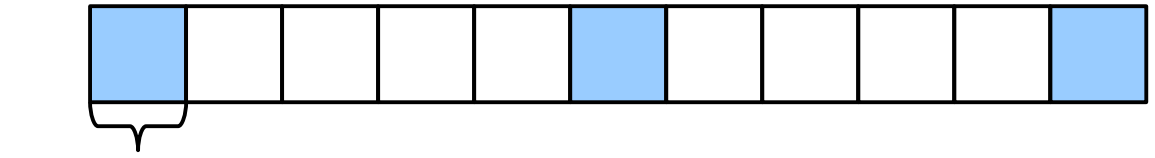
a[4][4]

```
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

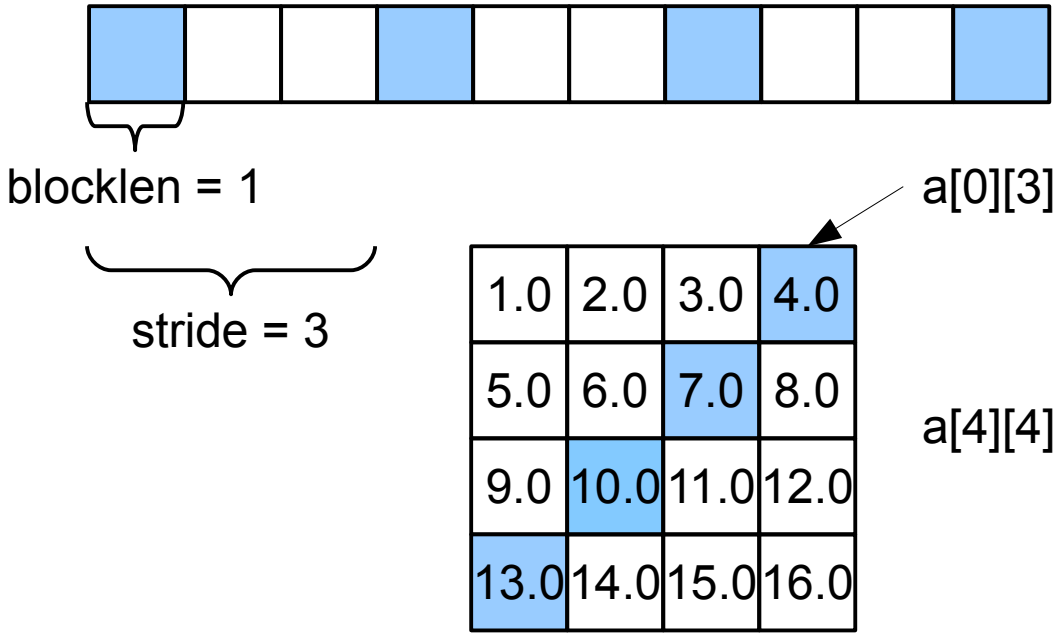
a[4][4]

```
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()

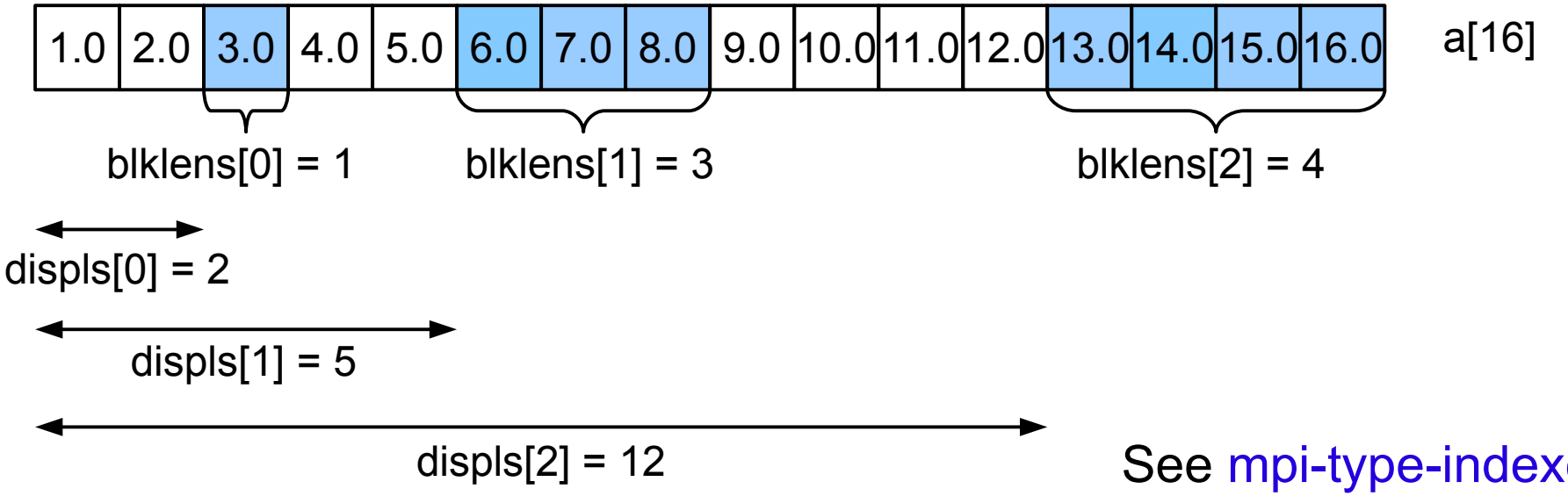
```
MPI_Datatype newtype;
MPI_Type_indexed( ... );
MPI_Type_commit(&newtype);
```

```
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

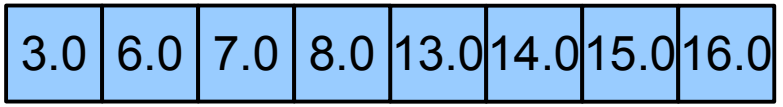
MPI_Type_indexed()

```
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);
```



See mpi-type-indexed.c

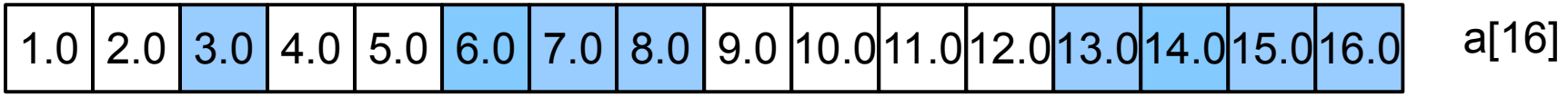
```
MPI_Send(&a[0], 1, newtype, dest, tag, MPI_COMM_WORLD);
```



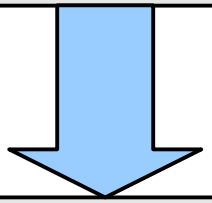
1 element of type newtype

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);
```



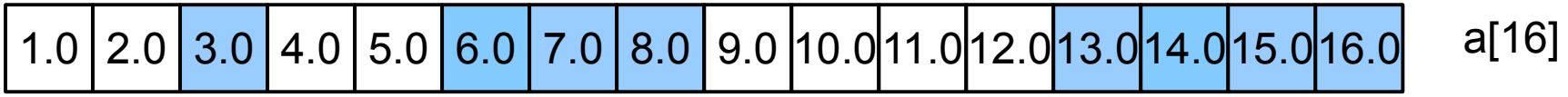

```
MPI_Send(&a[0], 1, newtype, dest, tag, MPI_COMM_WORLD);
```



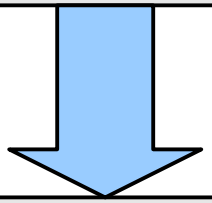
```
MPI_Recv(&b[0], 1, newtype, src, tag, MPI_COMM_WORLD);
```

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);
```




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



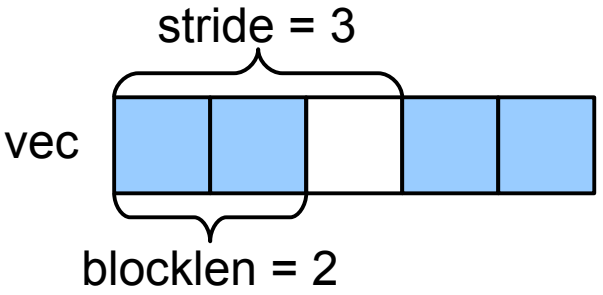
```
MPI_Recv(&b[0], 8, MPI_FLOAT, src, tag, MPI_COMM_WORLD);
```

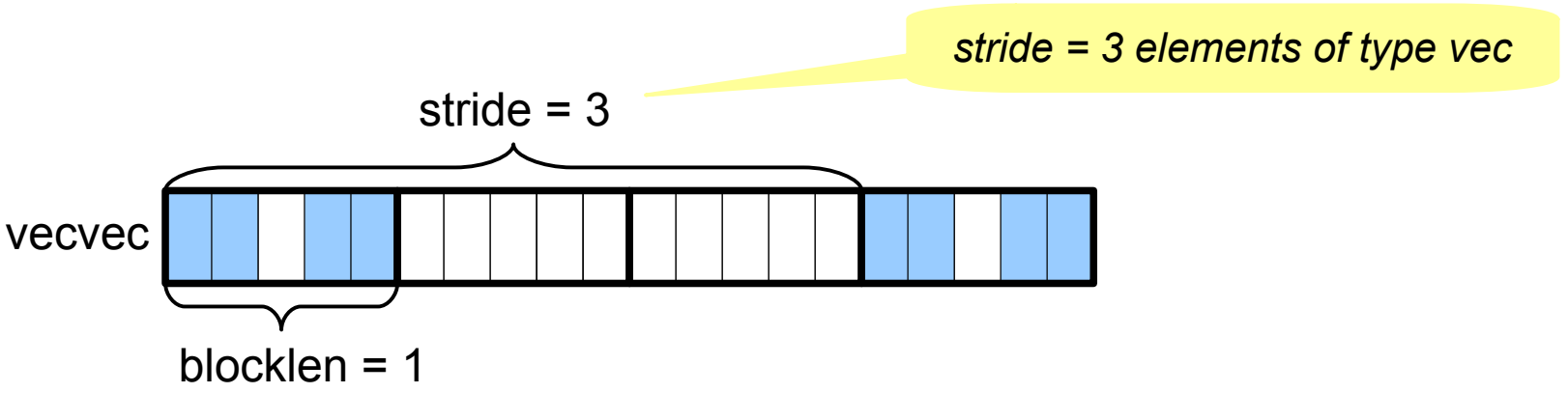
Combining custom datatypes

- 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);
```





See mpi-vecvec.c

MPI_Type_create_struct()

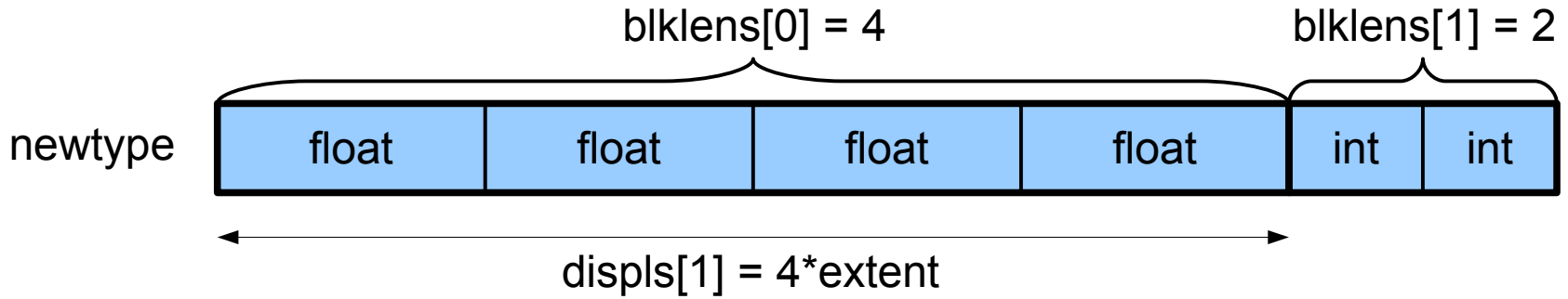
```
int MPI_Type_create_struct(int count, int
*array_of_blklen, MPI_Aint *array_of_displ,
MPI_Datatype *array_of_types, MPI_Datatype *newtype)
```

- 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_create_struct()

```
typedef struct {
    float x, y, z, v;
    int n, t;
} particle_t;

int count = 2; int blklens[2];
MPI_Aint displs[2], lb, extent;
MPI_Datatype oldtypes[2], newtype;

oldtypes[0] = MPI_FLOAT; blklens[0] = 4; displs[0] = 0;
MPI_Type_get_extent(MPI_FLOAT, &lb, &extent);
oldtypes[1] = MPI_INT; blklens[1] = 2; displs[1] = 4*extent;
MPI_Type_create_struct(count, blklens, displs, oldtypes, &newtype);
MPI_Type_commit(&newtype);
```

"Lower Bounds" are used to specify types that have "holes" at the beginning. See https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node74.htm#Node74 for details.



See mpi-type-struct.c

Concluding Remarks

- Message passing is a very basic 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