

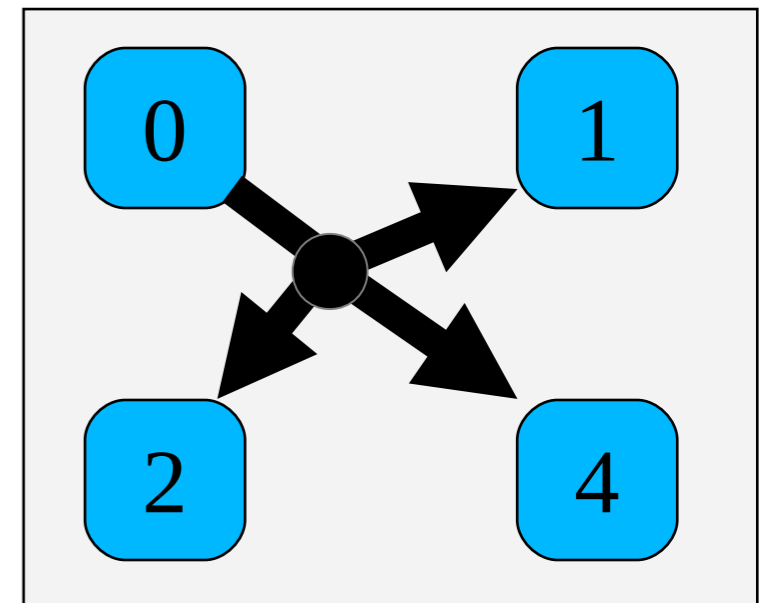
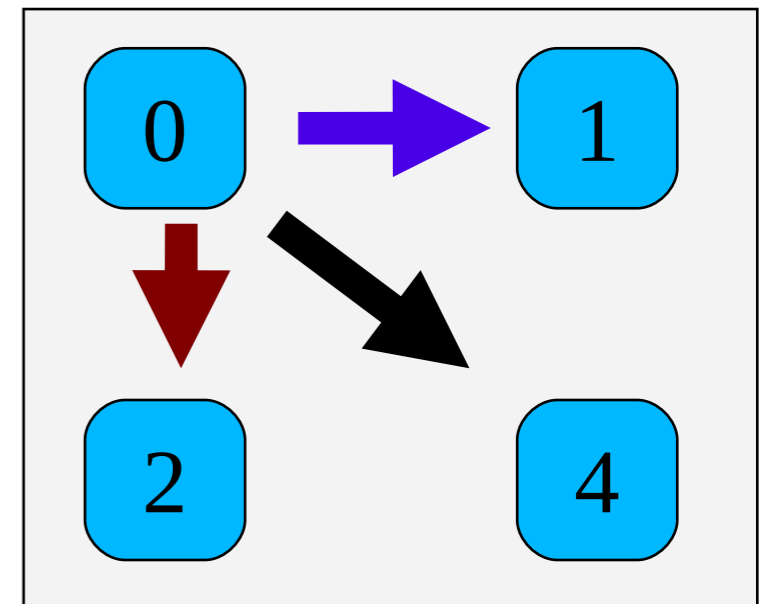


MPI point-to-point communication

Slides Sebastian von Alfthan

Introduction

- MPI processes are independent, they communicate to coordinate work
- **Point-to-point communication**
 - Messages are sent between two processes
- **Collective communication**
 - Involving a number of processes at the same time
 - More on this tomorrow!



MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match – one receive per send

MPI point-to-point operations

- Each message (envelope) contains
 - The actual **data** that is to be sent
 - The **datatype** of each element of data.
 - The **number of elements** the data consists of
 - An identification number for the message (**tag**)
 - The **ranks** of the the source and destination process

Send operation

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

- Parameters
 - **buf** The data that is sent
 - **count** Number of elements in buffer
 - **datatype** Type of each element in buf (see later slides)
 - **dest** The rank of the receiver
 - **tag** An integer identifying the message
 - **comm** Communicator
 - **error** The function returns an error value; in C/C++ it's the return value of the function, and in Fortran an additional output parameter

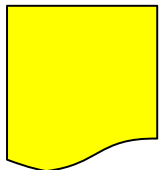
Send operation

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

- Special values for parameters
 - **dest**
 - **MPI_PROC_NULL** No operation takes place.
 - **comm**
 - **MPI_COMM_WORLD** Default communicator, this includes all processes
 - **error**
 - **MPI_SUCCESS** Return value if operation was successful.

Send operation

- C/C++ binding
 - `int MPI_Send(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - The return value of the function is the error value
- Fortran binding
 - `MPI_SEND(buffer, count, datatype, dest, tag, comm, ierror)`
 - `<type> buf(*)`
 - `integer count, datatype, dest, tag, comm, ierror`
 - **ierror** The error value



Receive operation

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

- Parameters
 - **buf** Buffer for storing received data.
 - **count** Number of elements in buffer, not the number of elements that are actually received.
 - **datatype** Type of each element in buf
 - **source** Sender of the message.
 - **tag** Number identifying the message.
 - **comm** Communicator
 - **status** Information on the received message
 - **error** As for send operation

Receive operation

`MPI_Recv(buf, count, datatype, source, tag, comm, status)`

- Special values for parameters
 - **source**
 - **MPI_ANY_SOURCE** Receive from any sender.
 - **MPI_PROC_NULL** No operation takes place.
 - **tag**
 - **MPI_ANY_TAG** Receive messages with any tag.
 - **comm**
 - **MPI_COMM_WORLD** Default communicator, the only one we use here
 - **status**
 - **MPI_IGNORE_STATUS** Do not store any status

Receive operation

- C/C++ binding

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

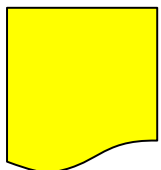
- Fortran binding

- `MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)`

- `<TYPE> BUF(*)`

- `INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM`

- `INTEGER STATUS(MPI_STATUS_SIZE), IERROR`

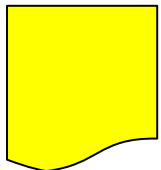


MPI datatypes

- MPI has a number of predefined datatypes to represent data
- Each C or Fortran datatype has a corresponding MPI datatype
 - C examples: **MPI_INT** for int and **MPI_DOUBLE** for double
 - Fortran example: **MPI_INTEGER** for integer
- MPI datatypes are not real types, cannot be used for variable definition

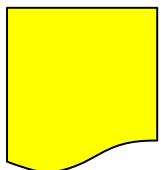
MPI datatypes

<u>MPI-type</u>	<u>C-type</u>
MPI_CHAR	signed char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	int long
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned int long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	

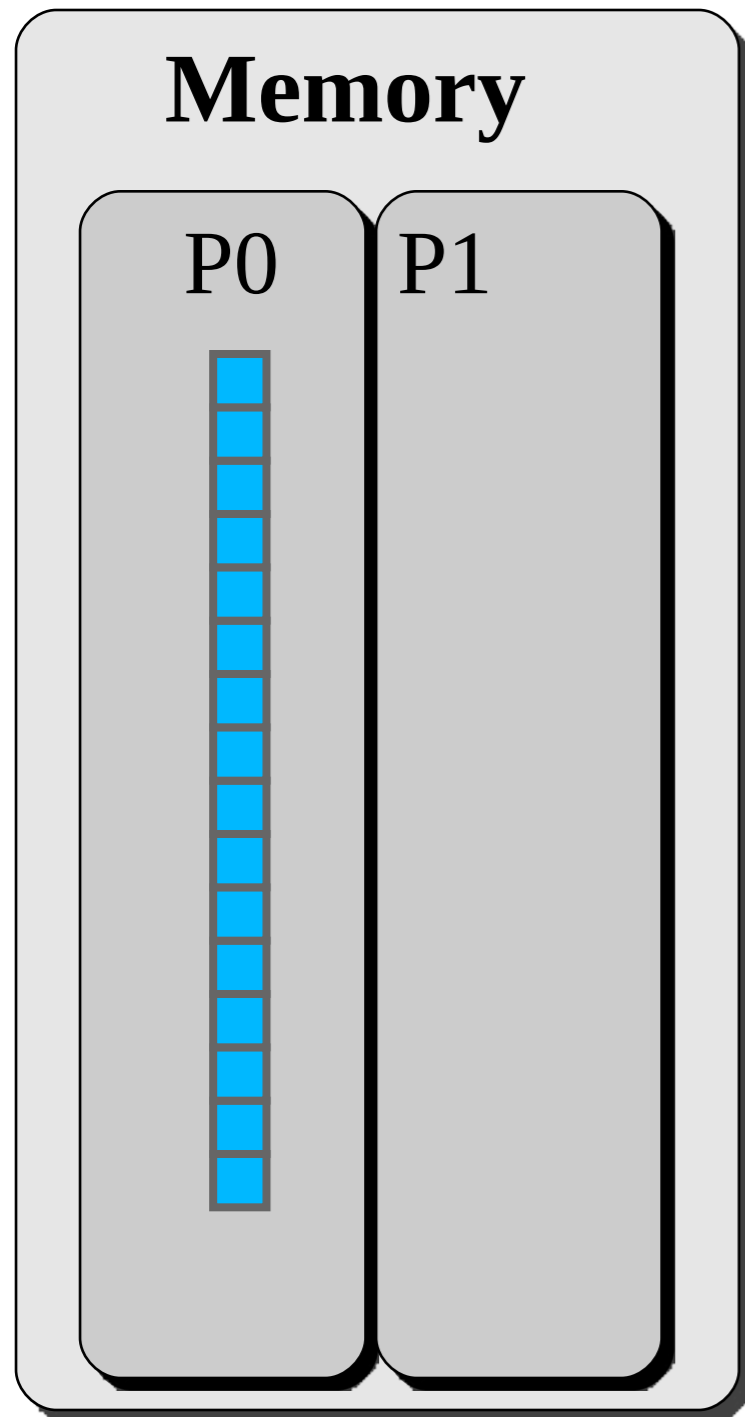


MPI datatypes

<u>MPI-type</u>	<u>Fortran-type</u>
MPI_CHARACTER	CHARACTER
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_REAL8	REAL*8 (non-standard)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	

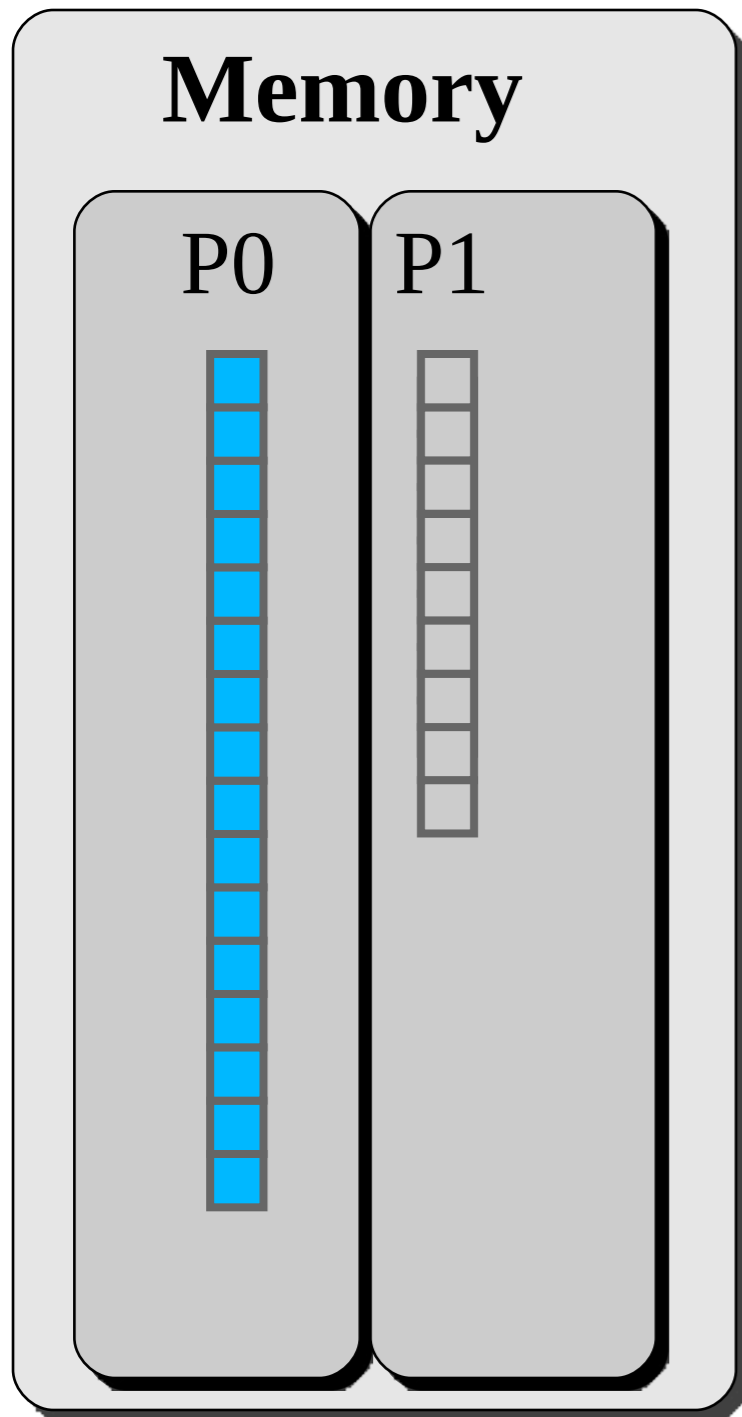


Case study 1: Parallel Sum

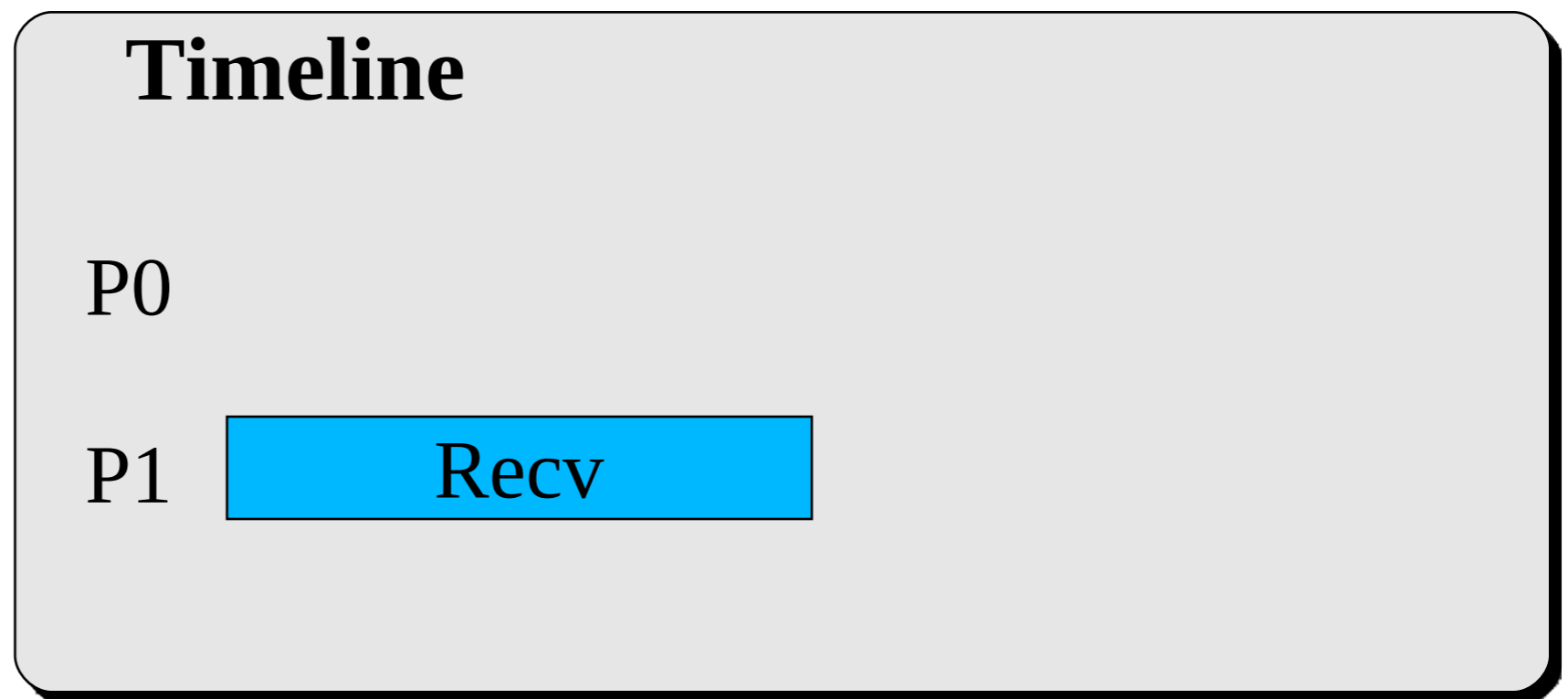


- Array is originally on process 0 (P0)
- Parallel algorithm
 - Scatter
 - Half of the array is sent to process 1
 - Compute
 - P0 & P1 sum independently their segment
 - Reduction
 - Partial sum on P1 sent to P0
 - P0 sums the partial sums

Case study 1: Parallel Sum

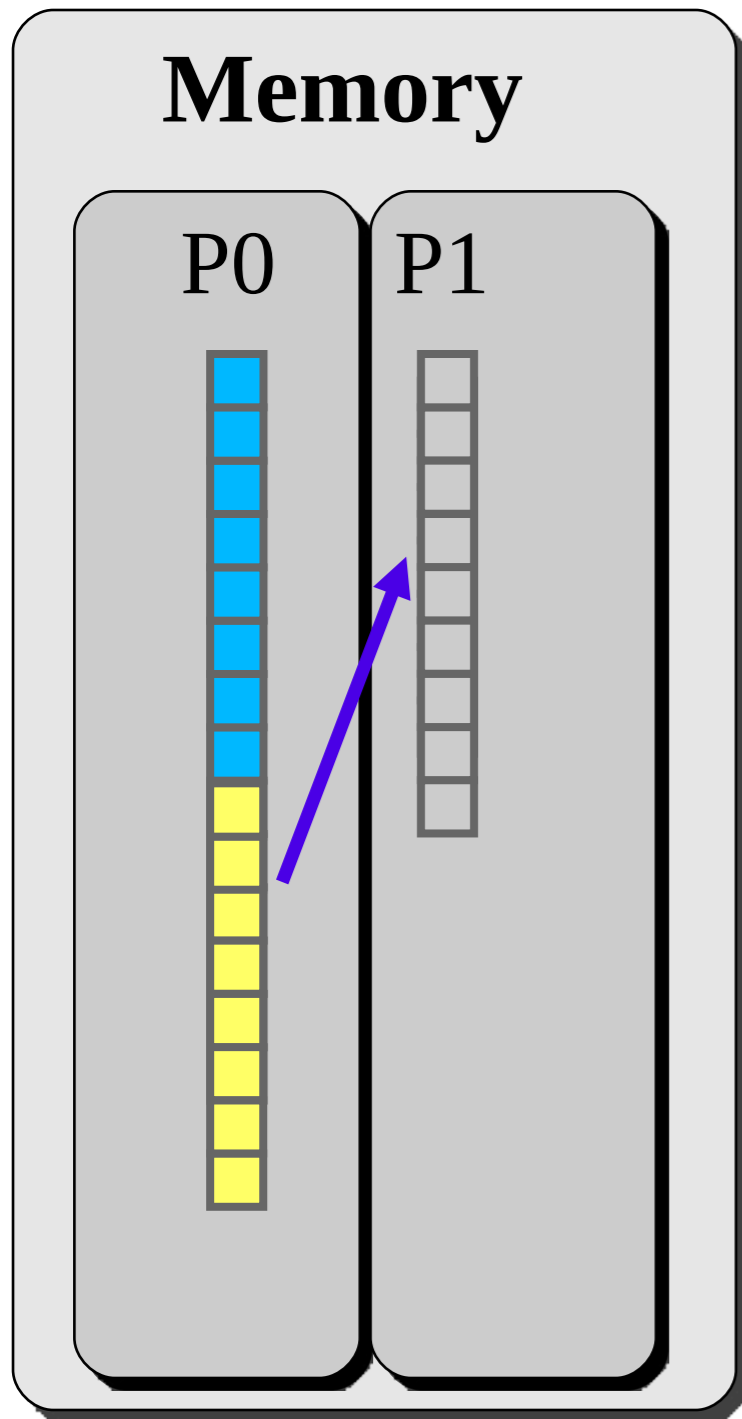


Step 1: Receive operation in scatter

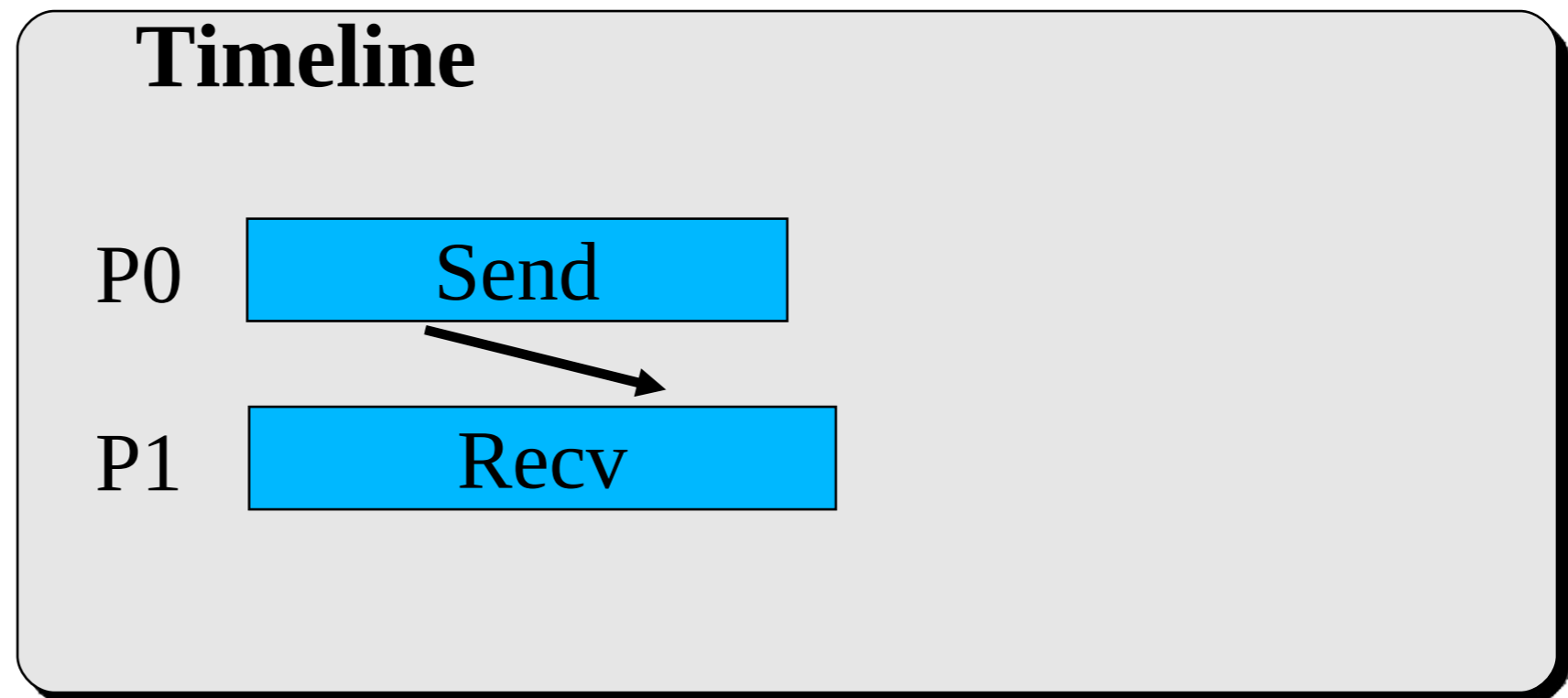


P1 posts a receive to receive half of the array from process 0

Case study 1: Parallel Sum

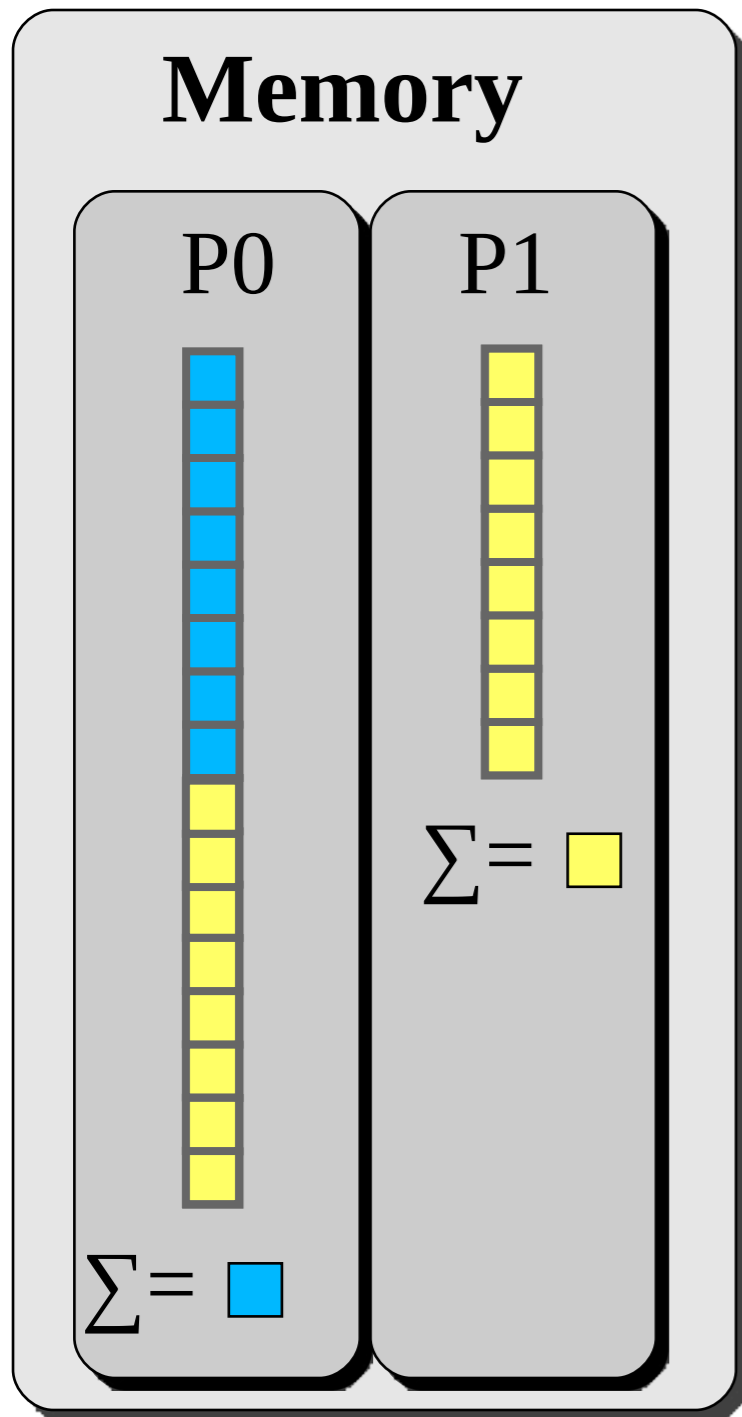


Step 2: Send operation in scatter

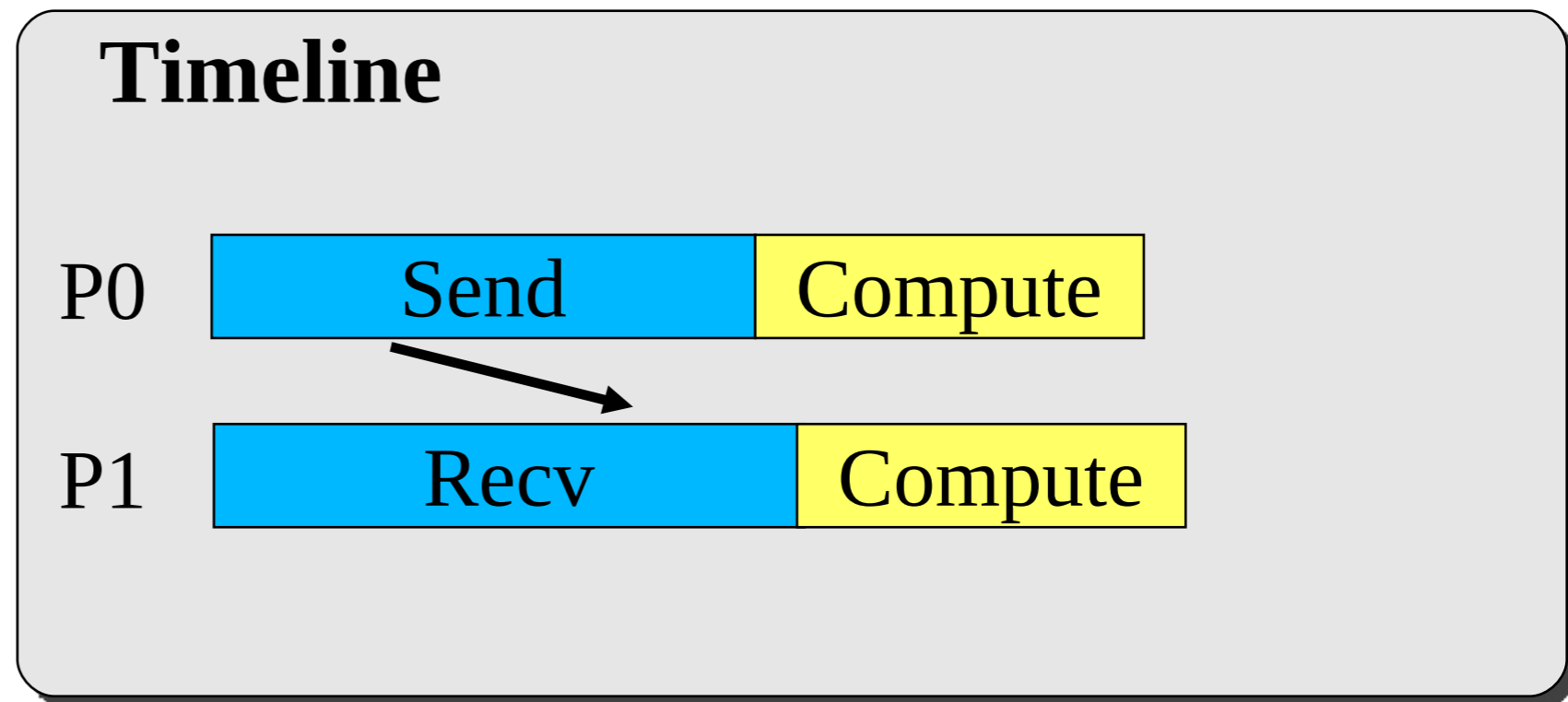


P0 posts a send to send the lower part of the array to P1

Case study 1: Parallel Sum

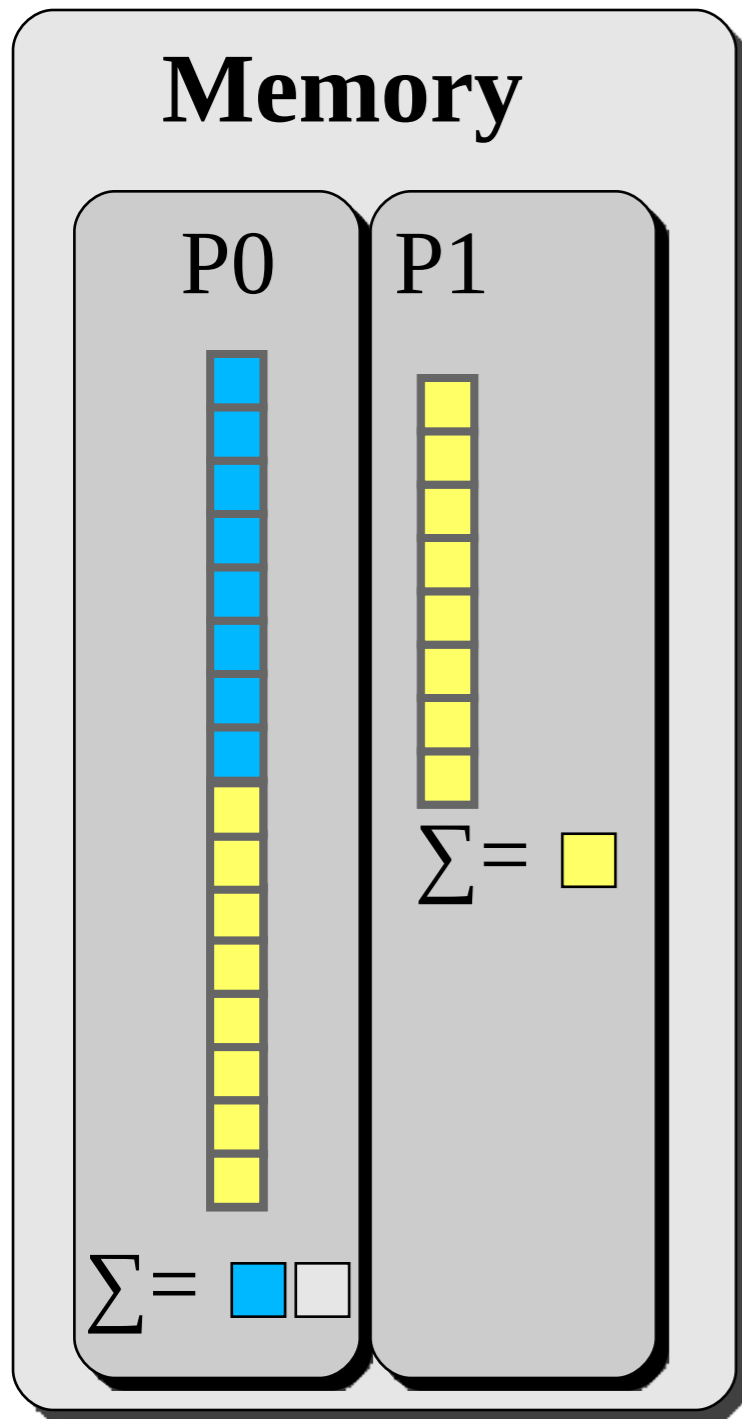


Step 3: Compute the sum in parallel

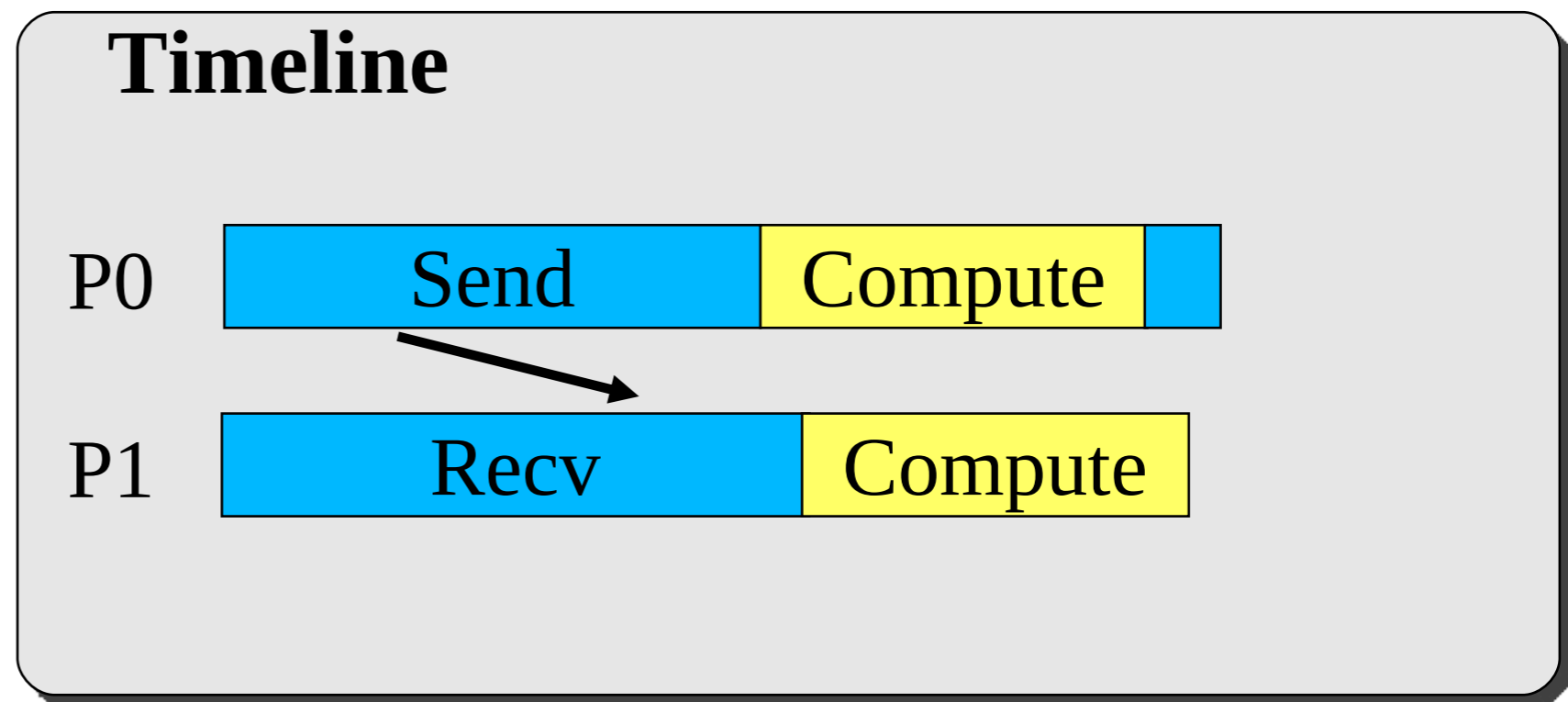


P0 & P1 computes their partial sums and store them locally

Case study 1: Parallel Sum

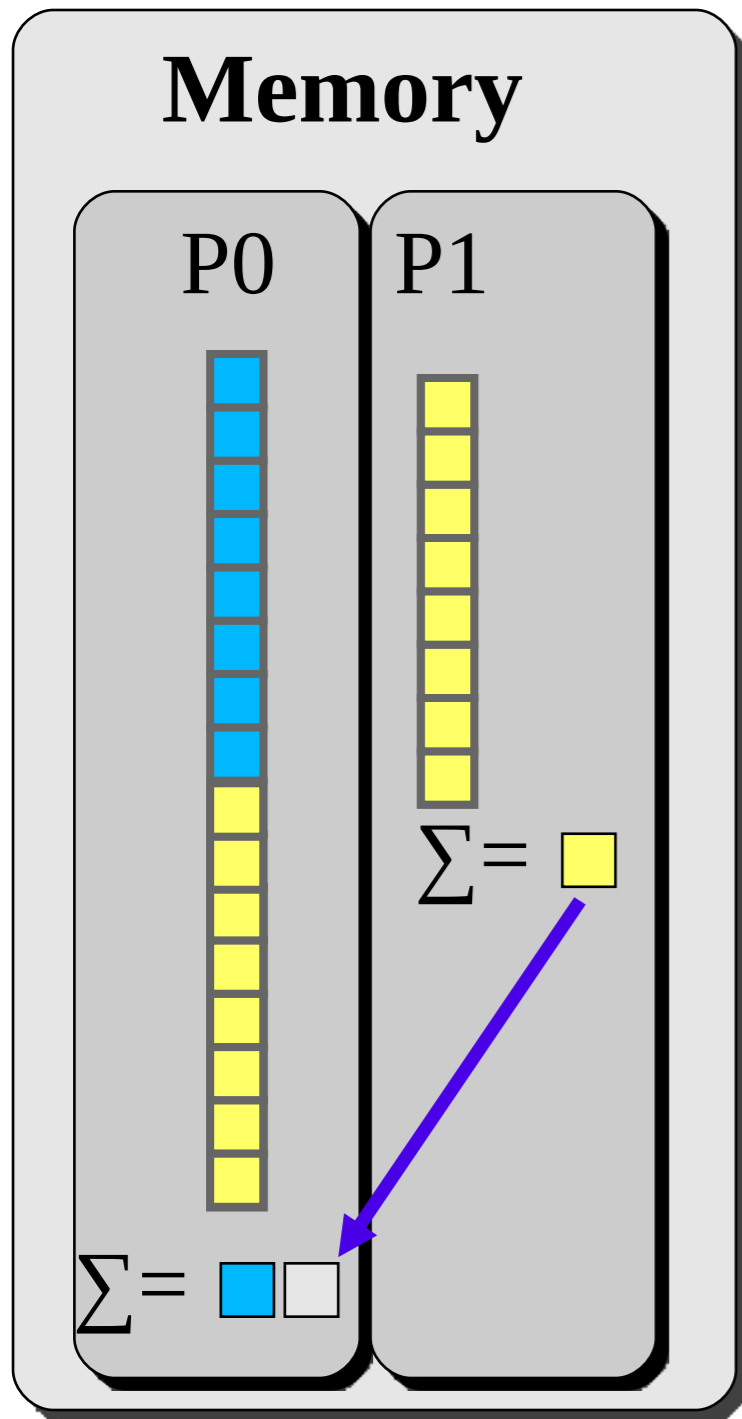


Step 4: Receive operation in reduction

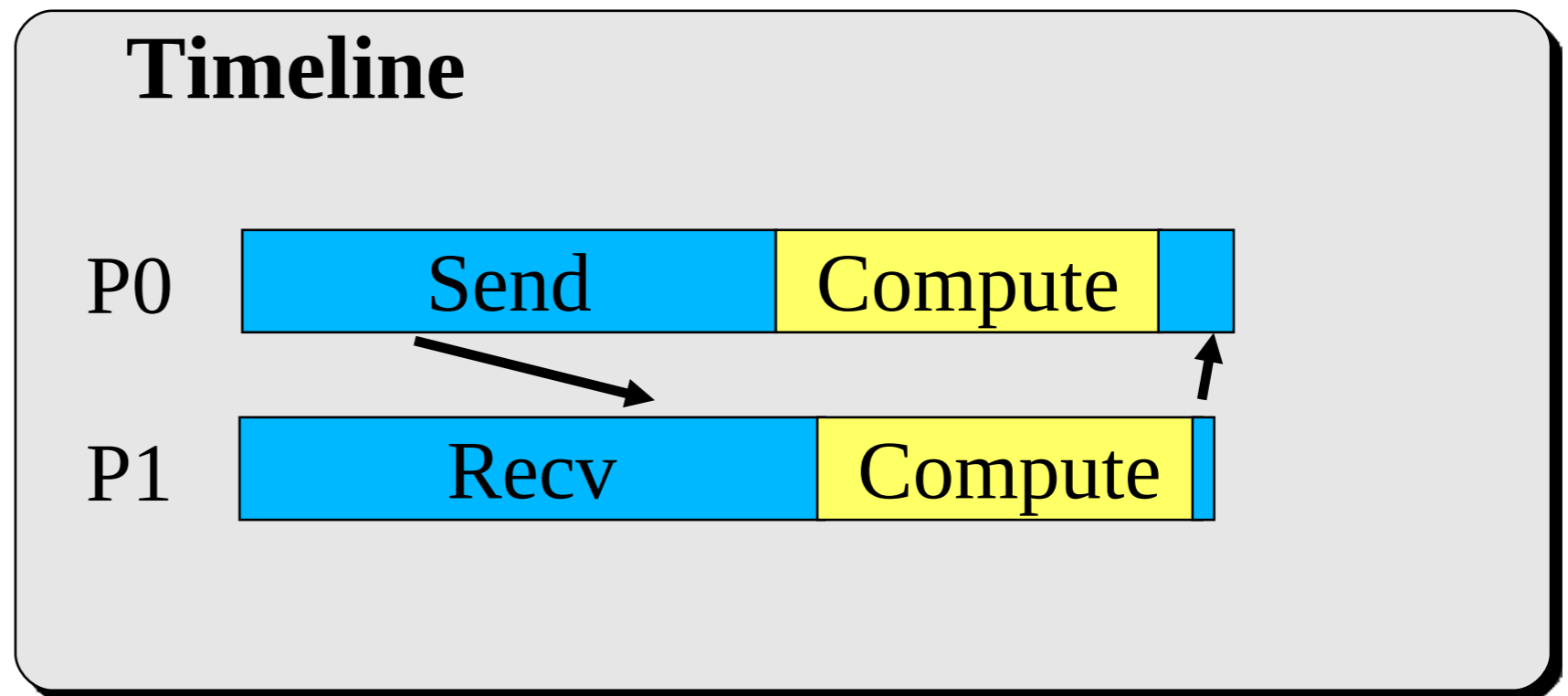


P0 posts a receive to receive partial sum

Case study 1: Parallel Sum

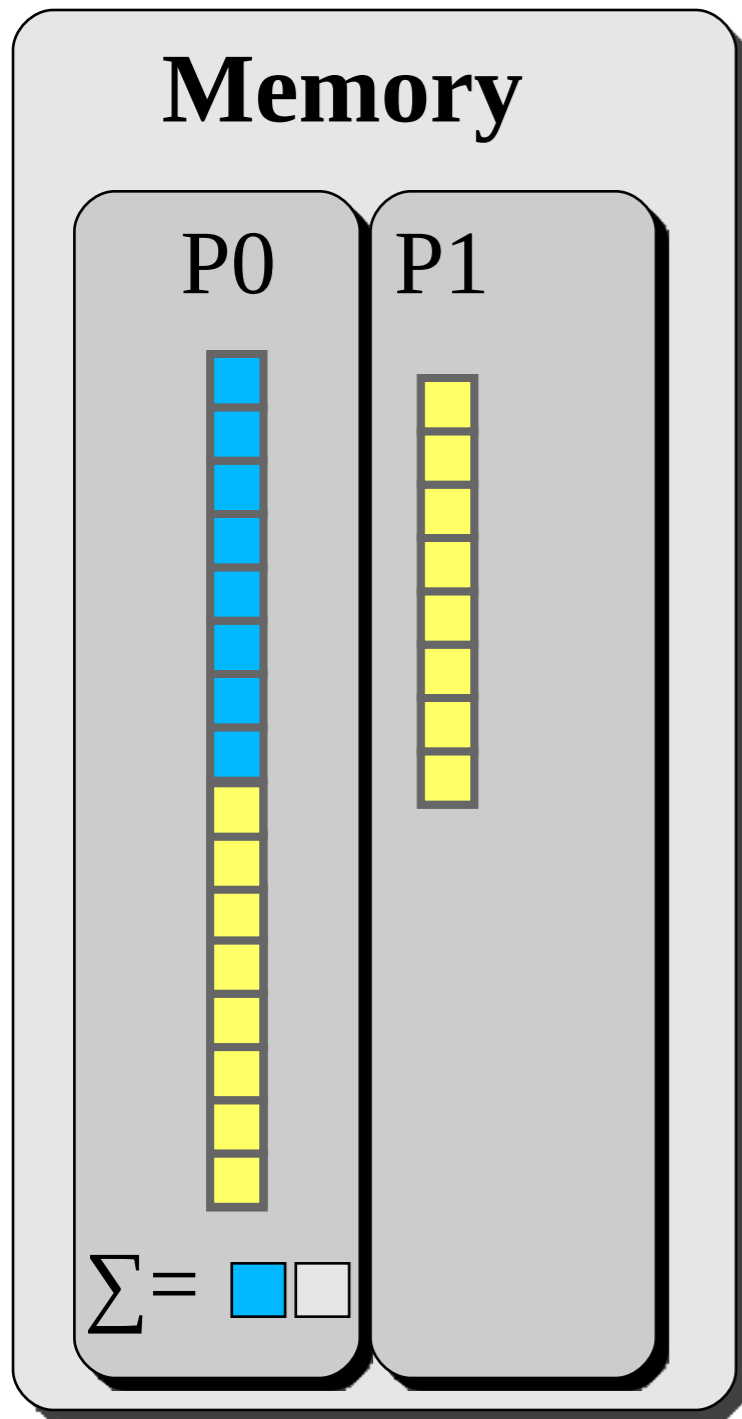


Step 5: Send operation in reduction

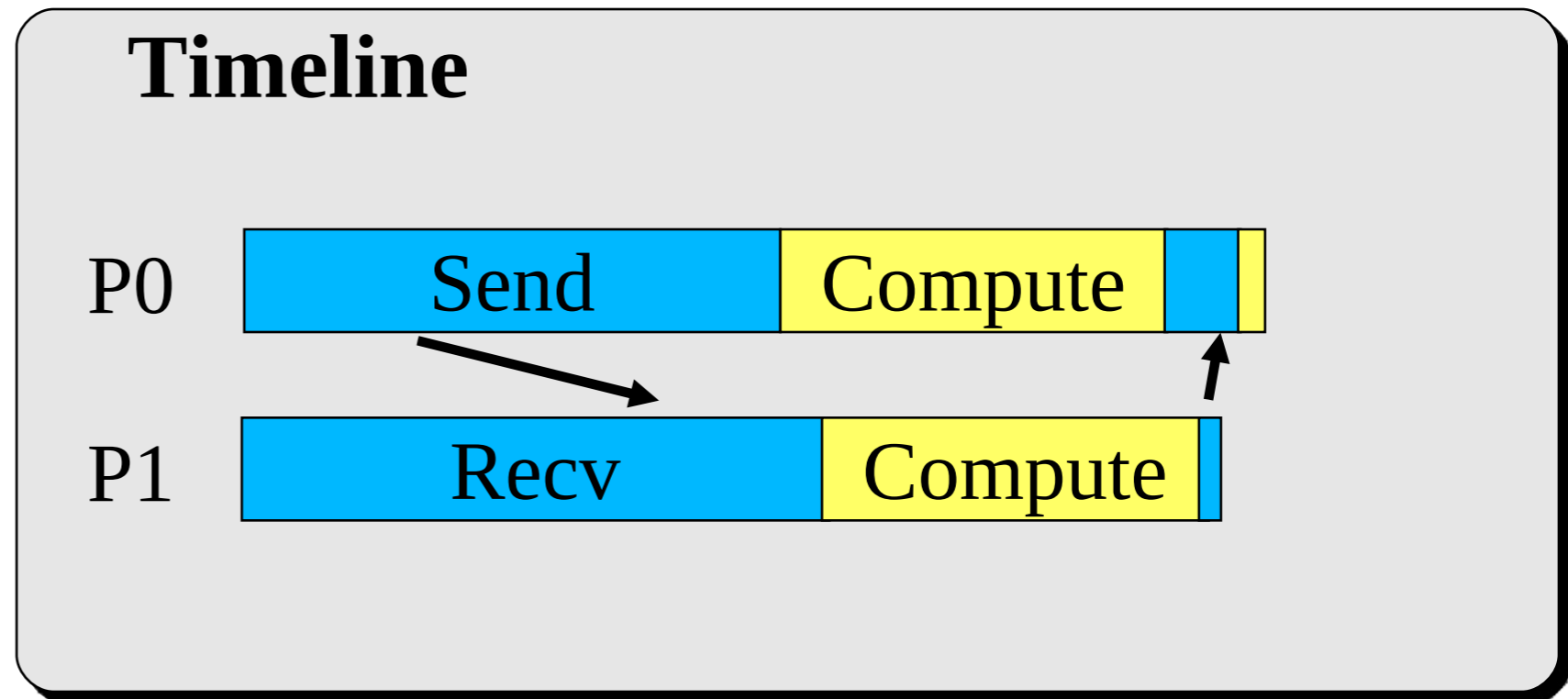


P1 posts a send with partial sum

Case study 1: Parallel Sum



Step 6: Compute final answer



P0 sums the partial sums

Demonstration— parallel sum

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

int main(int argc, char *argv[]){
    int i,N;
    double *array;
    double sum;
    N=100;
    array=malloc(sizeof(double)*N);

    for(i=0;i<N;i++){
        array[i]=1.0;
    }

    sum=0;
    for(i=0;i<N;i++){
        sum+=array[i];
    }
    printf("Sum is %g\n",sum);
}
```

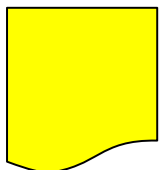
- We will parallelize this piece of code
- Let us begin...

Status parameter

- The status parameter in MPI_Recv contains information on how the receive succeeded
 1. Number of received elements
 2. Tag of the received message
 3. Rank of the sender
 4. And more...

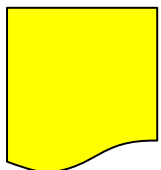
Status parameter: C/C++

- **Datatype** of status
 - Structure of type MPI_Status
- 1. Number of received elements**
 - Use the MPI_Get_count function to extract this information
 - `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
 - status return status of receive operation
 - datatype datatype of each receive buffer element
 - count number of received elements
- 1. Tag of received message**
 - `status.MPI_TAG`
- 1. Rank of the sender**
 - `status.MPI_SOURCE`



Fortran: Status parameter of receives

- **Datatype** of status
 - Integer array of size `MPI_STATUS_SIZE`
- 1. Number of received elements**
 - Use the `MPI_GET_COUNT` function to extract this information
 - `MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)`
`INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR`
 - `status` Return status of receive operation
 - `datatype` datatype of each receive buffer element
 - `count` number of received elements
 - `ierror` return value
- 1. Tag of received message**
 - `STATUS(MPI_TAG)`
- 1. Rank of the sender**
 - `STATUS(MPI_SOURCE)`



Demo – parallel sum while checking status

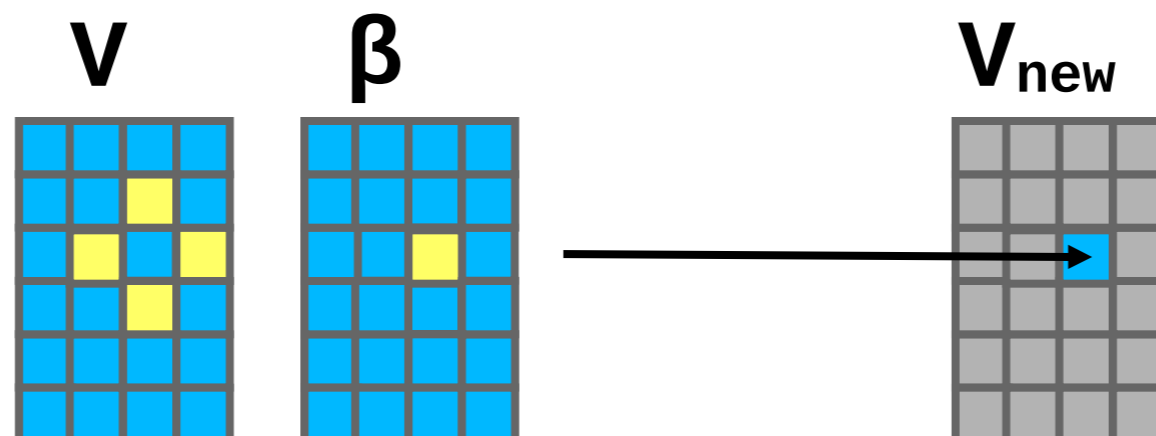
- We will continue to work on the parallel code we developed and use the status to check the size of the received message
- Let us begin...

Blocking routines & deadlocks

- Blocking routines
 - Completion depends on other processes
 - Risk for deadlocks – the program is stuck forever
- MPI_Send
 - Exits once the send buffer can be safely read and written to
- MPI_Recv
 - Exits once it has received the message in the receive buffer

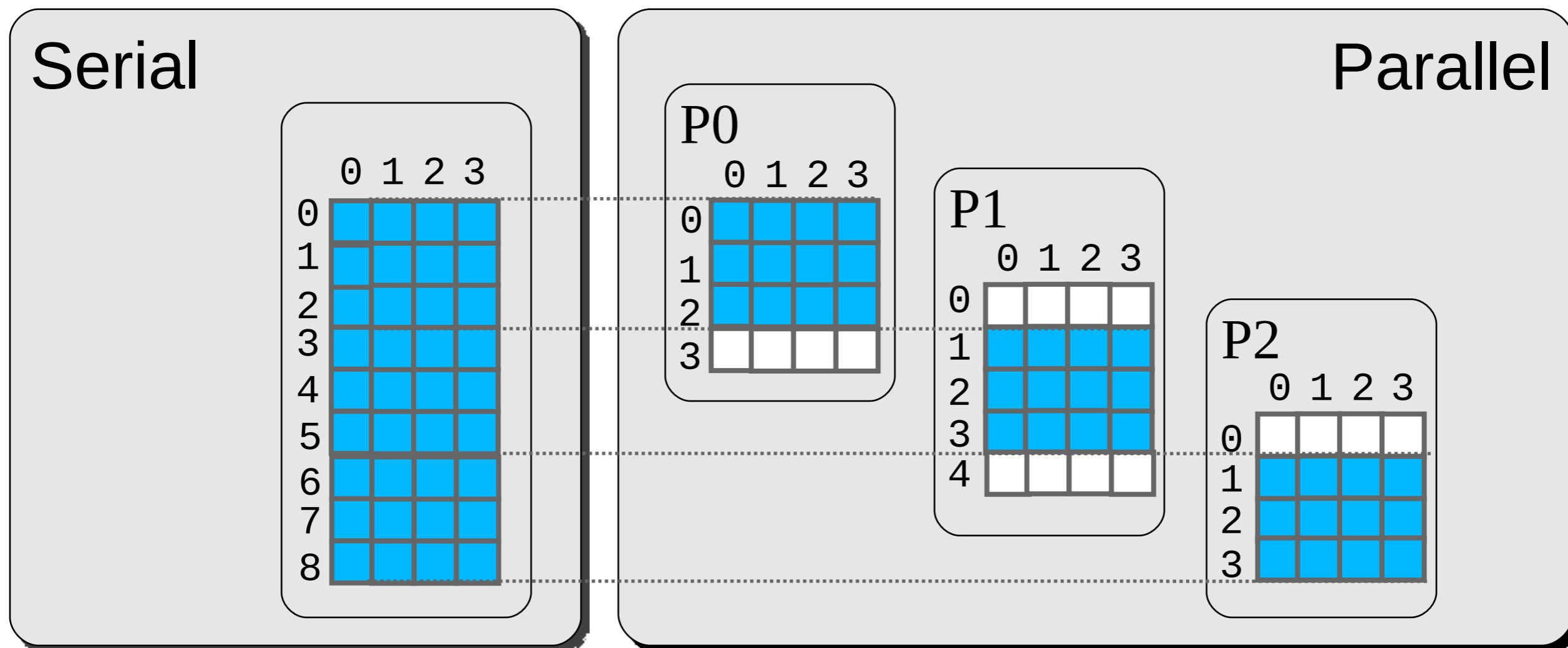
Case study 2: Jacobi solver

- Iteratively update the value of a 2D-array V
 - $V_{\text{new}}(i, j) = [V(i-1, j) + V(i+1, j) + V(i, j-1) + V(i, j+1) - \beta(i, j)] / 4$
- This is called a Jacobi iterator and is a way of solving Poisson's equation $\nabla^2 V = \beta$ using an iterative scheme



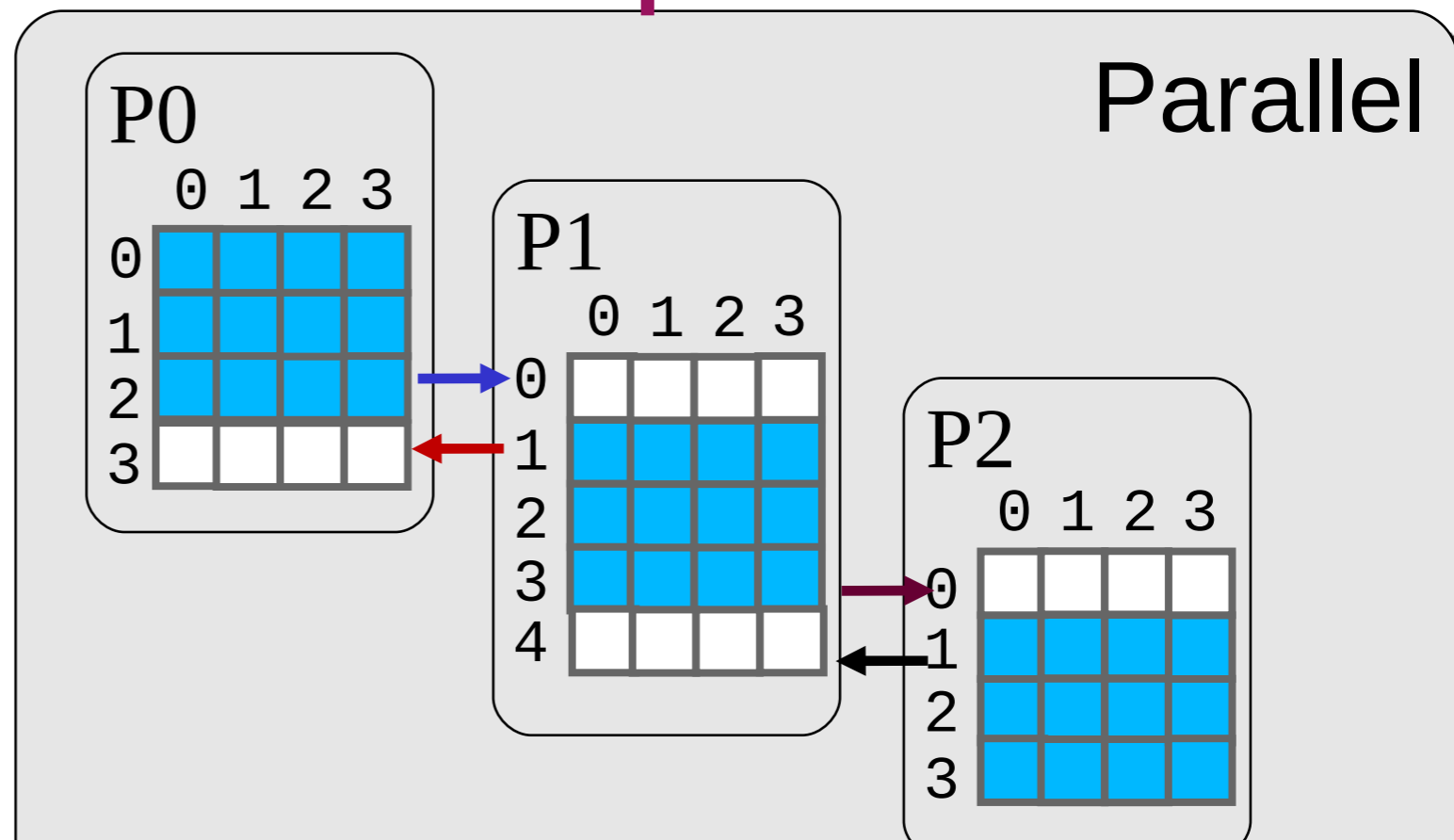
Cs2: Domain decomposition

- Divide the area in domains to solve it in parallel
- Ghost layer at boundary represent the value of the elements of the other process

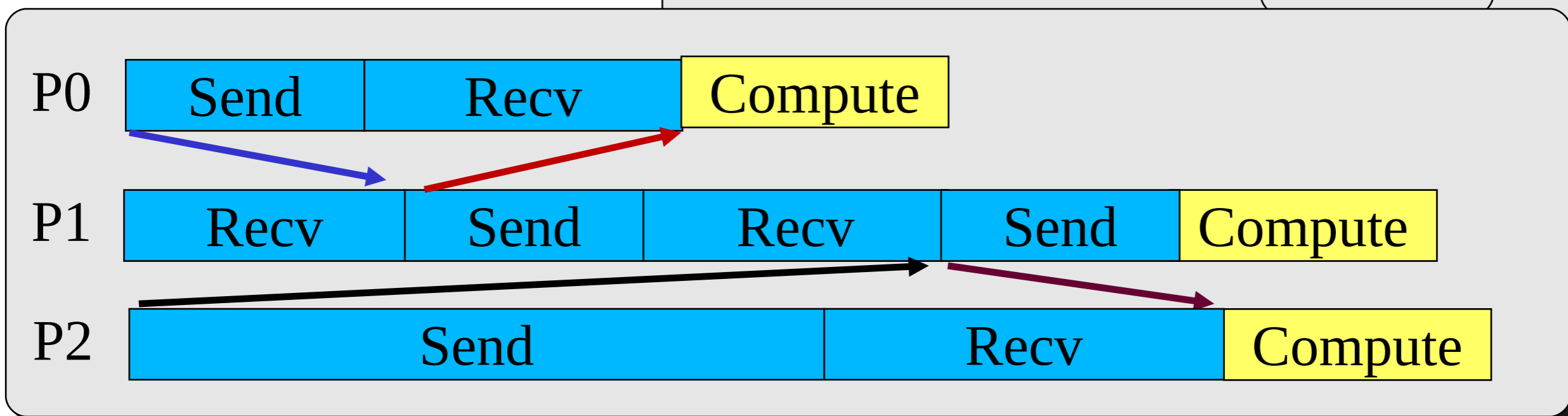


CS2: One iteration step

- Need to carefully schedule the order of sends and receives to avoid deadlocks



Timeline



MPI_Send MPI_Recv code

```
...
ngbr_up=rank-1;
ngbr_down=rank+1;
if(ngbr_up<0) ngbr_up=MPI_PROC_NULL;
if(ngbr_down>=nprocs) ngbr_down=MPI_PROC_NULL;

for(i=0;i<max_iterations;i++){
    if(rank%2==0){
        MPI_Send(V[1],M,MPI_DOUBLE,ngbr_up,10,MPI_COMM_WORLD);
        MPI_Recv(V[0],M,MPI_DOUBLE,ngbr_up,11,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Send(V[N],M,MPI_DOUBLE,ngbr_down,12,MPI_COMM_WORLD);
        MPI_Recv(V[N+1],M,MPI_DOUBLE,ngbr_down,13,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    if(rank%2==1){
        MPI_Recv(V[N+1],M,MPI_DOUBLE,ngbr_down,10,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Send(V[N],M,MPI_DOUBLE,ngbr_down,11,MPI_COMM_WORLD);
        MPI_Recv(V[0],M,MPI_DOUBLE,ngbr_up,12,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Send(V[1],M,MPI_DOUBLE,ngbr_up,13,MPI_COMM_WORLD);
    }
    compute(V,Vn,beta,N,M);
    SWAP(V,Vn,dpptemp);
}
...
```

Combined send & receive

`MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag,
comm, status)`

- Parameters
 - As for MPI_Send and MPI_Recv combined
- Sends one message and receives another one, with one single command
 - Reduces risk for deadlocks
- Destination rank and source rank can be different, or same
- Usage examples:
 - Two process exchange data with each other
 - Pipe or ring of processes exchanging data

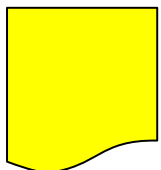
Combined send & receive

- C/C++ binding

- `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)`

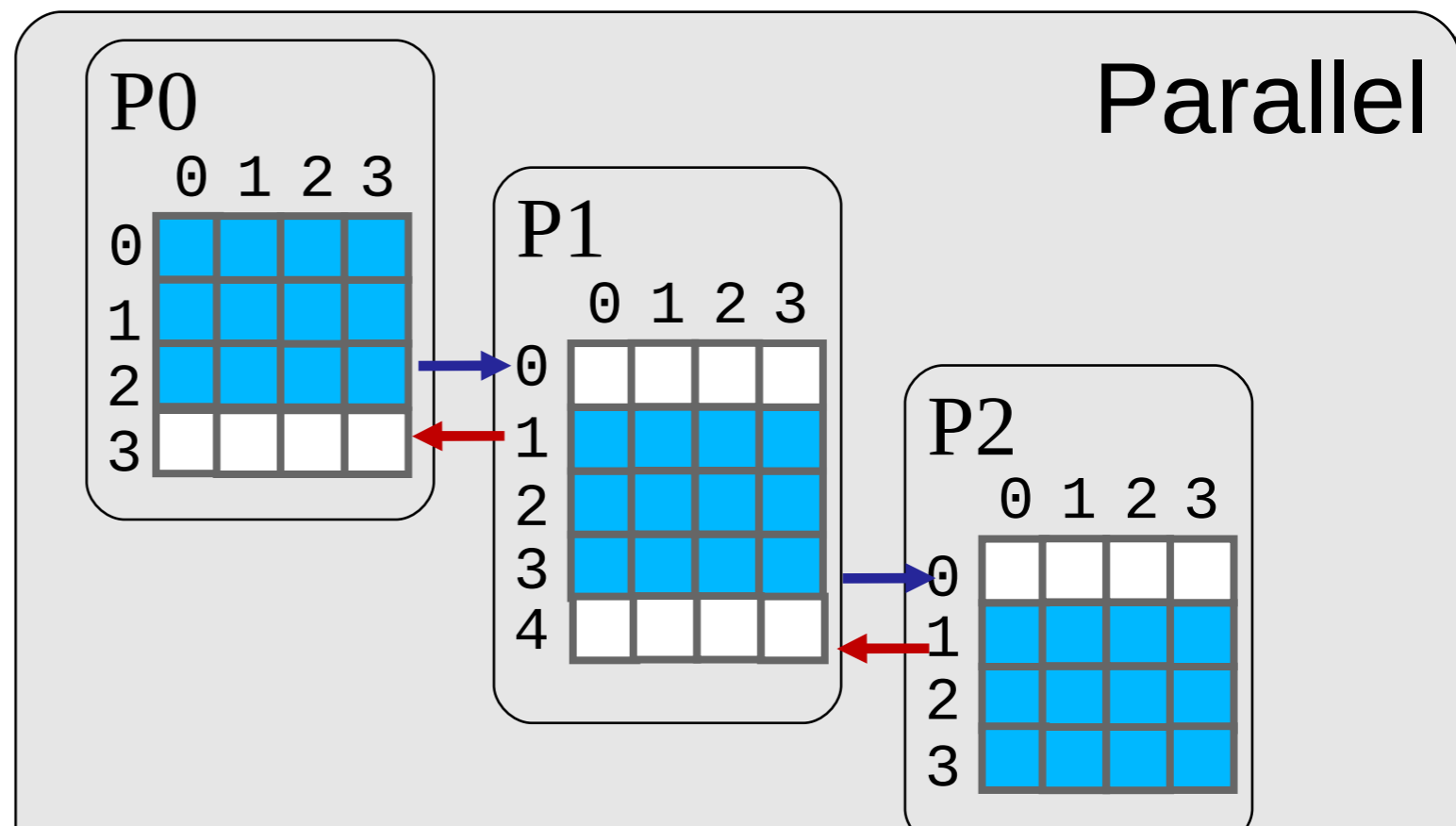
- Fortran binding

- `MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF, RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)`
- `<TYPE> SENDBUF(*), RECVBUF(*)`
- `INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE, SOURCE, RECV TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR`

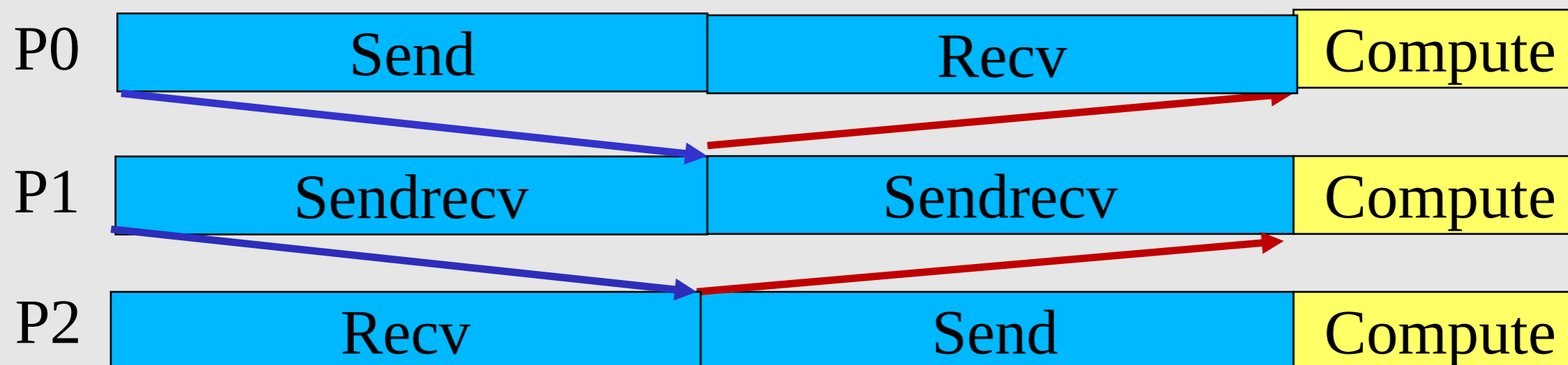


Cs2: MPI_sendrecv

- MPI_Sendrecv
 - Sends and receives with one command
 - No risk of deadlocks



Timeline



Demonstration: Halo exchange with MPI_sendrecv



- We will rewrite the algorithm using MPI_Sendrecv
 - This allows us to make it much more simple!
- Let us begin...

Summary

- Point-to-point communication
 - Messages are sent between two processes
- We discussed send and receive operations enabling any parallel application
 1. Blocking: MPI_Send MPI_Recv
 2. Blocking: MPI_Sendrecv
- Tomorrow more sends and receives
 1. Non-blocking: MPI_Isend MPI_Irecv
- Collective routines, discussed tomorrow, will make some things easier