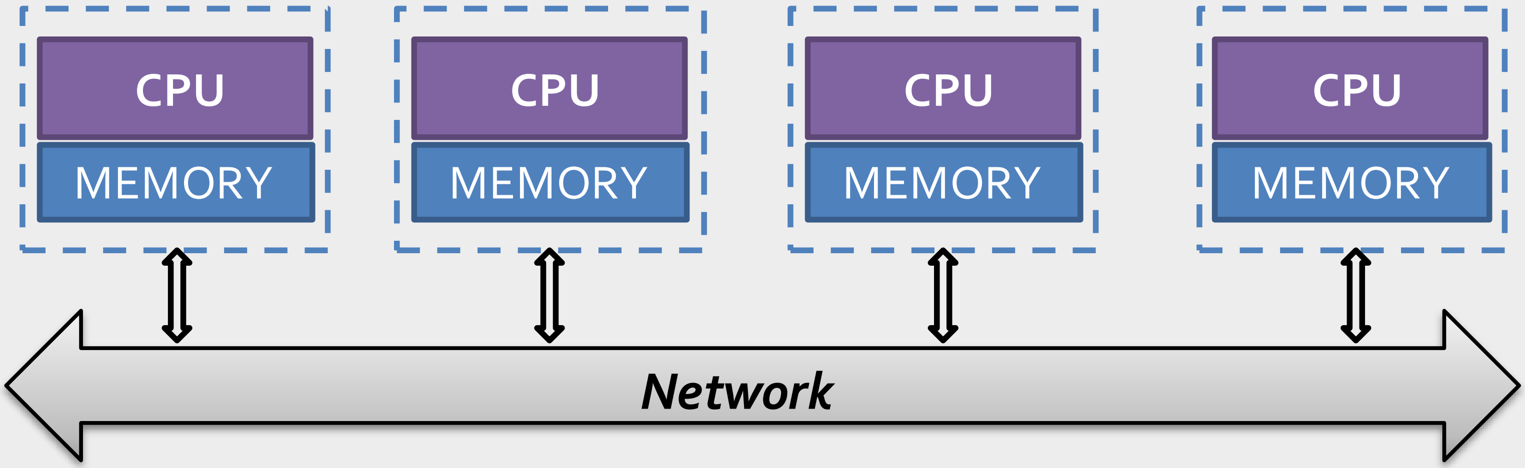
Background

- **Parallel Computer Memory Architectures**

Distributed Memory | Shared Memory | Hybrid Shared-Distributed Memory



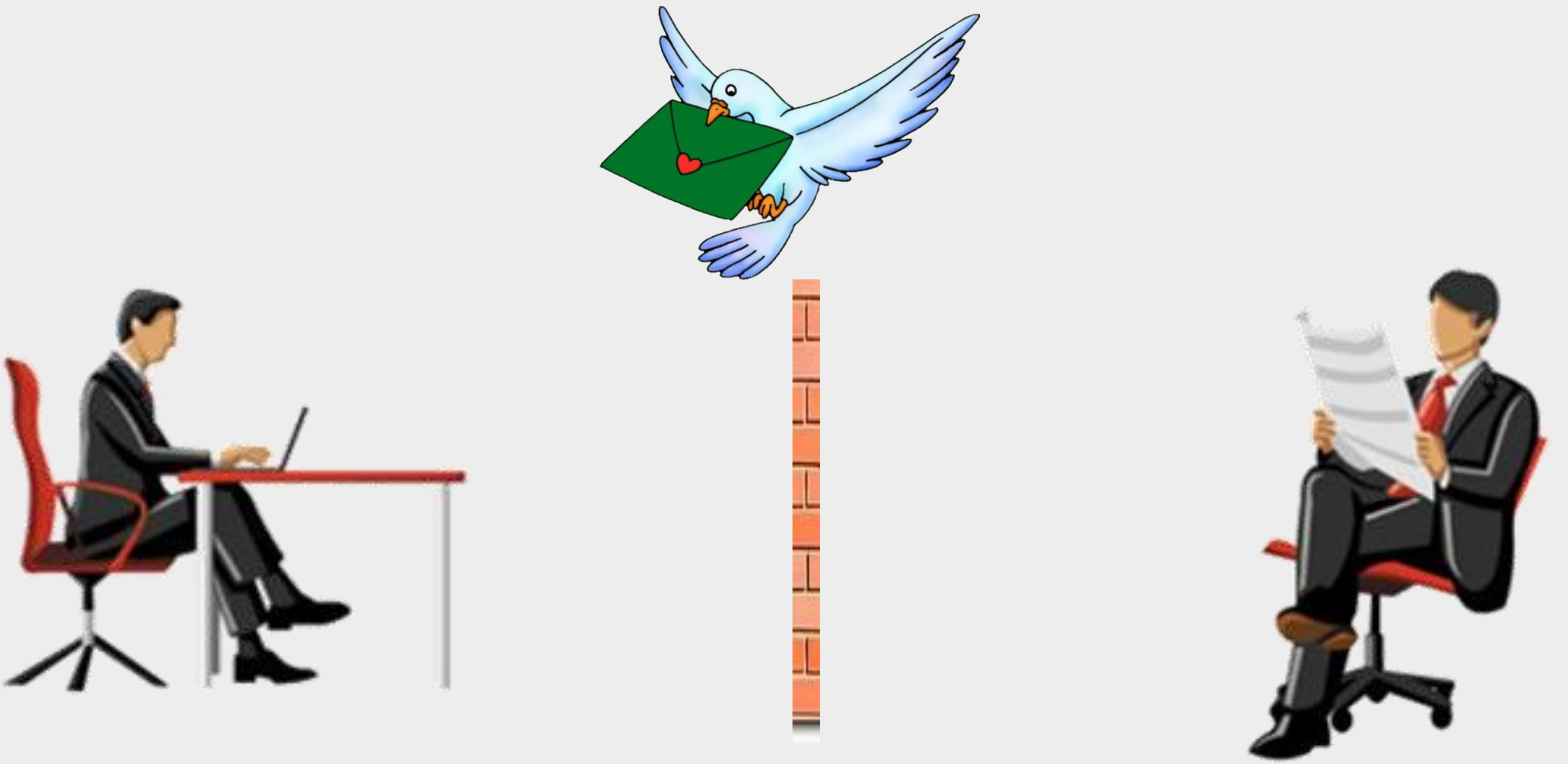
MPP (*massively parallel processor*)

Could we design PP with OpenMP for Distributed Memory Systems?

Shared vs Distributed Memory

Shared vs Distributed Memory

Distributed Memory



ITMO UNIVERSITY

Parallel Programming with MPI

Katerina Bolgova, PhD
eScience Research Institute & HPC Department

MPI Overview

M P I = **Message Passing Interface**:

- is a ***specification*** for the developers and users of message passing libraries.
- By itself, it is NOT a library - but rather the specification of what such a library should be.

- MPI's goals are high performance, scalability, and portability
- Most MPI implementations consist of a specific set of routines
- All parallelism is explicit: the programmer is responsible for identifying parallelism correctly and implementing parallel algorithms using MPI constructs.
- Usually MPI programs are designed with **SPMD-pattern.**

MPI Overview

- There are many implementations of MPI (MPICH, LAM/MPI, WMPI, Intel MPI, Open MPI, MPJ, Mvapich)
- There are implementations of mpi-libraries for different programming languages such as Fortran, C/C++, python and Java

Brief history:

1992: The beginning of MPI design.

Nov 1993: draft MPI standard presented.

May 1994: Final version of MPI-1.0 released

Jun 1995: MPI-1.1
Jul 1997: MPI-1.2
Sep 2008: version MPI-2.1 released
Sep 2009: MPI-2.2
Sep 2012: The MPI-3.0 standard was approved.

June 2015: MPI-3.1 was approved.

- ***Documentation for all versions of the MPI standard is available at: http://www.mpi-forum.org/docs/.***

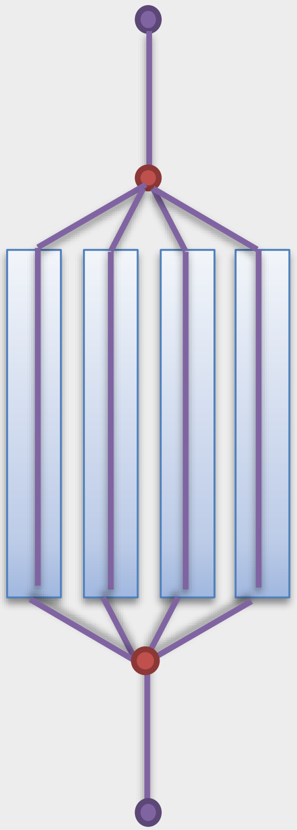
MPI Concepts

- There is **NO** shared memory (no shared variables)
- Each process has its own address space
- Each process has its own ***Rank***
- All *interprocess communications are performed* through sending/receiving messages

The basic concepts of MPI standard:

- Data types
- Communication operations
- Communication contexts and process groups
- Process topologies

General MPI Program Structure



- Programs that make MPI library calls required ***MPI include file***
- ***Initialization*** of MPI environment (parallel code starts here). Could appear only once in a program.
- Each process has its own address space for calculations. Message passing calls are used for inter-process communication .
- ***Termination*** of MPI environment (parallel code ends here). Could appear only once in a program.

Data Types

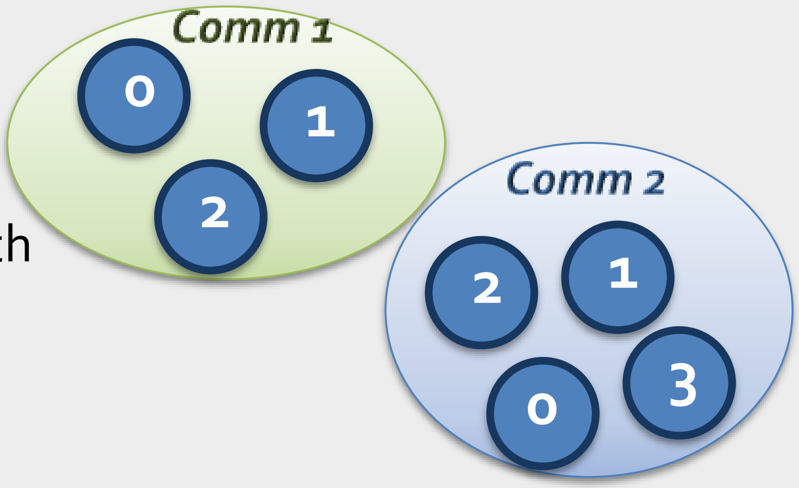
MPI predefines its elementary data types:

- **MPI_BYTE** and **MPI_PACKED** do not correspond to standard C or Fortran types
- Programmers may also create their own data types

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Communicators Overview

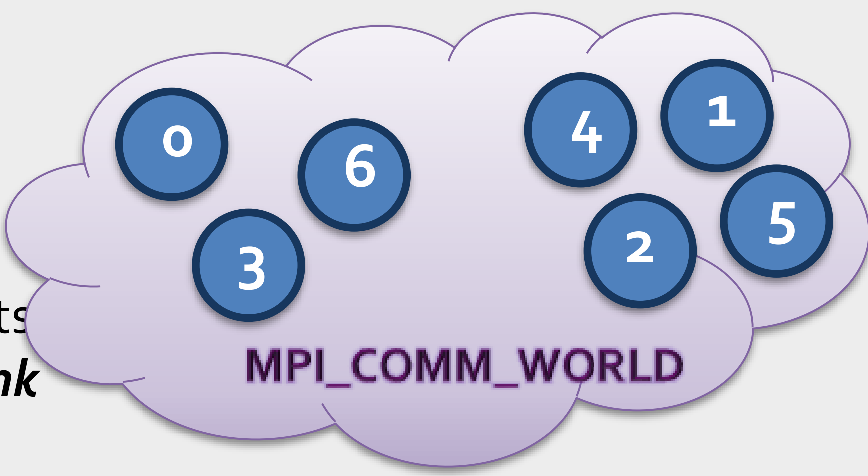
- Communicator objects connect groups of processes in the MPI session



- Communicators are used to define which collection of processes may communicate with each other

- Most MPI routines require specifying a communicator as an argument

- **MPI_COMM_WORLD** is the predefined communicator that includes all of your MPI processes



- Within a communicator, every process has its unique, integer identifier which is called ***Rank***

Initialization of MPI environment

- *Initialization of the parallel code :*

```
int MPI_Init ( int *agrc, char ***argv )
```

- *This function shows whether MPI_Init has been called :*

```
MPI_Initialized (&flag)
```

- *Definition of **the total number** of MPI processes in the specified communicator:*

```
int MPI_Comm_size ( MPI_Comm comm, int *size )
```

- *Definition of **the rank** of the called MPI process within specified communicator:*

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

Termination of MPI environment

- *Termination of the parallel code :*

```
int MPI_Finalize (void)
```

- *Time in Seconds since a fixed point in the past :*

```
double MPI_Wtime(void)
```

- *Definition of the Program Execution's Time*

```
double t_1, t_2, dt;
t_1 = MPI_Wtime();
...
t_2 = MPI_Wtime();

dt = t2 - t1;
```

General Scheme of MPI Programs

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

if ( ProcRank == 0 )
   DoManagerProcess();
else
   DoWorkerProcesses();
```

or

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 )
      DoProcess0();
else if ( ProcRank == 1 )
      DoProcess1();
else if ( ProcRank == 2 )
      DoProcess2();
             . . .
```

SPMD-pattern

Message passing

All inter-process communications are performed through sending/receiving messages

The communication operations

Point-to-Point	Collective
• typically involve message passing between **two** specific processes • basic point-to-point communication operations are **'send'** and **'receive'** • particularly useful in patterned or irregular communications	• must involve **all** processes within the scope of a communicator • types of collective operations: synchronization, data movement, collective computation • used only with MPI predefined data types

Point to Point Communications

Two main operations:

- Send message of ***type*** with ***count*** length from buffer ***buf*** to process with ***dest*** Rank:

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

- Receive message of ***type*** with ***count*** length to buffer ***buf*** from process with ***source*** Rank:

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

- **Buffer:** Application address space that references the data that is to be sent or received.
- **Count:** Indicates the number of data elements of a particular type.
- **Data Type:** MPI predefined elementary data types or derived data types.
- **Destination**: Specified as the rank of the ***receiving*** process
- **Source:** Specified as the rank of the ***sending*** process. This may be set to the wild card *MPI_ANY_SOURCE* to receive a message from *any task*.
- **Tag:** Unique identifier of a message. The wild card *MPI_ANY_TAG* *(just for 'receive')* is used to receive any message regardless of its tag.
- **Communicator:** Indicates the communication context (*MPI_COMM_WORLD* is usually used).
- **Status:** indicates the source of the message and the tag of the message. *(just for receive)*

1st MPI-program

```
#include "mpi.h"

int main(int argc, char* argv[]) {
  int ProcNum, ProcRank, RecvRank;
  MPI_Status Status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  if ( ProcRank == 0 ) {            // Process with Rank=0
     printf ("\n Hello from process %3d", ProcRank);

      for ( int i=1; i < ProcNum; i++ ) {
         MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
                            MPI_ANY_TAG, MPI_COMM_WORLD, &Status);

          printf("\n Hello from process %3d", RecvRank);
      }
  }
  else                              // All other processes
    MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);

  MPI_Finalize();                   // terminate parallel block
  return 0;
 }
```

Receive Messages

Process characteristics

```
int MPI_Recv( *buf, count, type, source, tag, comm, *status)
```

Message characteristics

- The buffer memory should be sufficient to receive the message
- Elements' types of sending and receiving messages must be identical
- When memory is insufficient, some parts of the message will be lost
- The Status parameter is a pointer to a predefined structure MPI_Status that helps to determine:
  - `status.MPI_SOURCE` – ***source*** of the message
  - `status.MPI_TAG` – ***tag*** of the message
  - length of the received message by calling a routine:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype type,
                  int *count )
```

Receive Messages

- For instance:

```
    ...
MPI_Status status;
int count;
MPI_Recv( ... ,  MPI_INT, ... , &status );
MPI_Get_count( &status, MPI_INT, &count );
/* ... in a variable count is the length of message */
```

Do we need to define a message length after it's been received?

To get into the ***status*** parameter information about the structure of the expected message:

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
                                          MPI_Status *status)
```

- The function returns a filled structure MPI_Status
- After that it's possible to find out the message length by MPI_Get_count calls

Point to Point Communications

There are different types of send and receive routines used for different purposes:

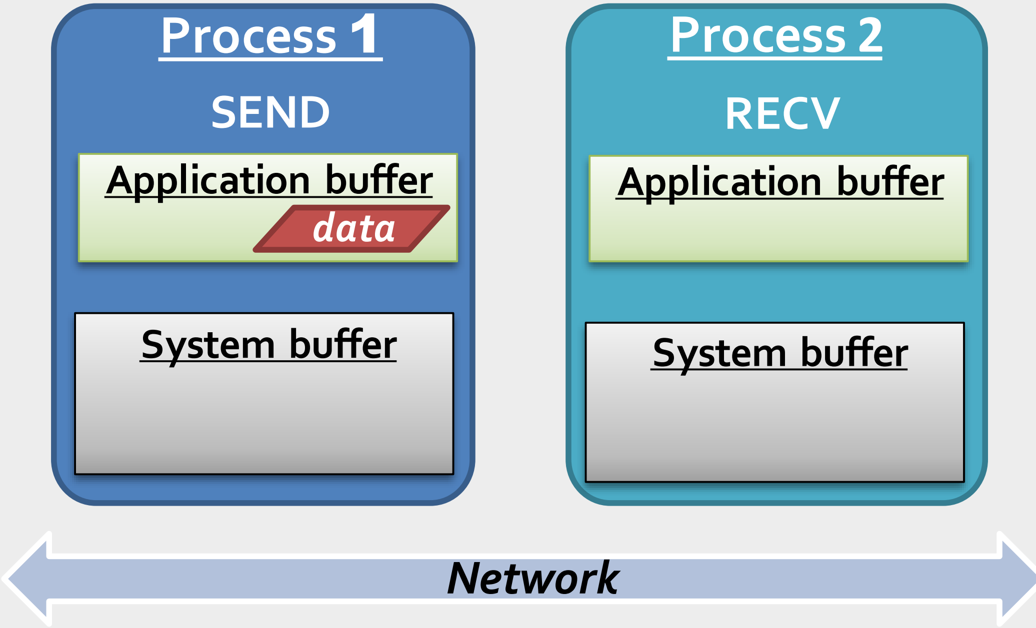
- Blocking communications
- "Ready" send
- Non-blocking communications
- Combined send/receive

Buffering in MPI

Consider the following two cases:

1. A send operation occurs 5 seconds before the receive is ready:
   where is the message while the receive is pending?

2. Multiple sends arrive at the same receiving task which can only accept one send at a time:
   what happens to the messages that are "backing up"?



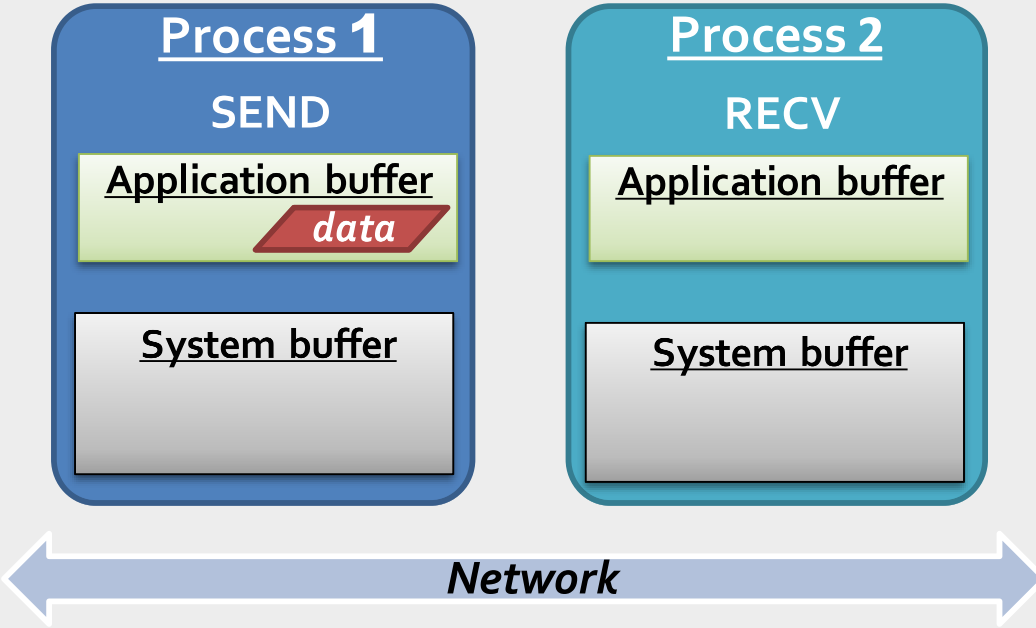
Buffering in MPI

Consider the following two cases:

1. A send operation occurs 5 seconds before the receive is ready:
   where is the message while the receive is pending?

2. Multiple sends arrive at the same receiving task which can only accept one send at a time:
   what happens to the messages that are "backing up"?

For instance, Scenario 1:
AppB1 -> SysB1 -> NW ->
-> SysB2 -> AppB2



Buffering in MPI

Consider the following two cases:

1. A send operation occurs 5 seconds before the receive is ready:

   where is the message while the receive is pending?

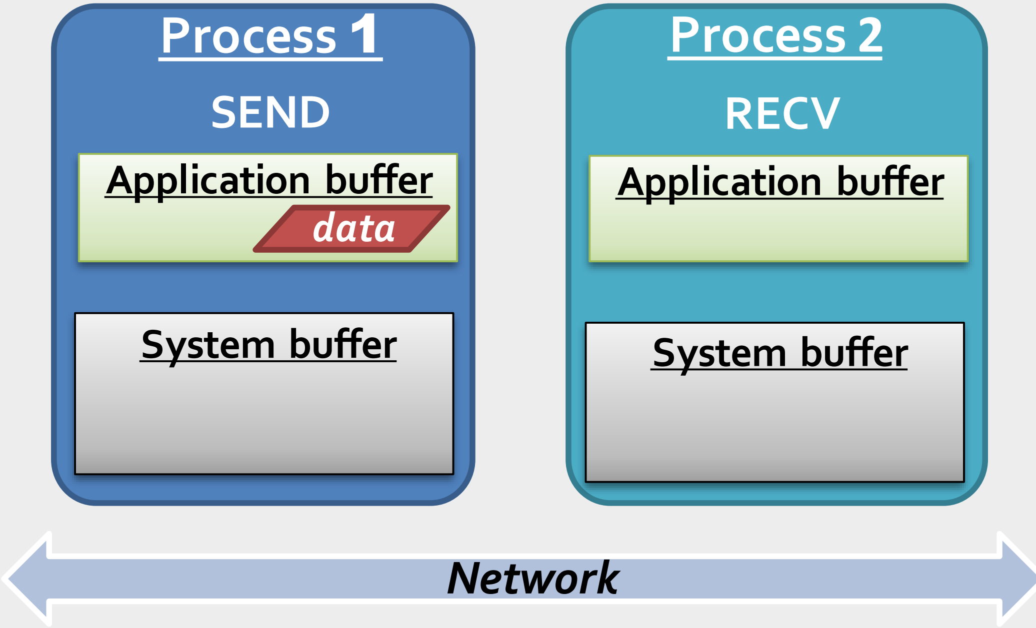
2. Multiple sends arrive at the same receiving task which can only accept one send at a time:

   what happens to the messages that are "backing up"?

For instance, Scenario 2:

AppB1 -> NW ->

-> SysB2 -> AppB2



Buffering in MPI

Consider the following two cases:

1. A send operation occurs 5 seconds before the receive is ready:
   where is the message while the receive is pending?

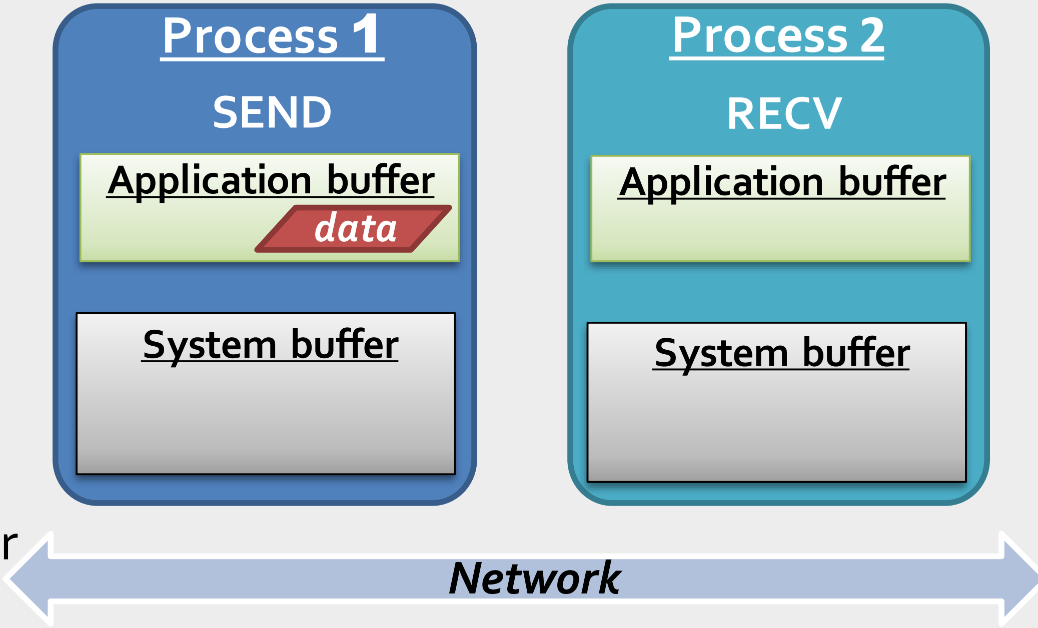
2. Multiple sends arrive at the same receiving task which can only accept one send at a time:
   what happens to the messages that are "backing up"?

System buffer space is:

- managed entirely by the MPI-lib
- a finite resource
- often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both.

Application buffer:

- User managed address space (i.e. your program variables)



Blocking operations

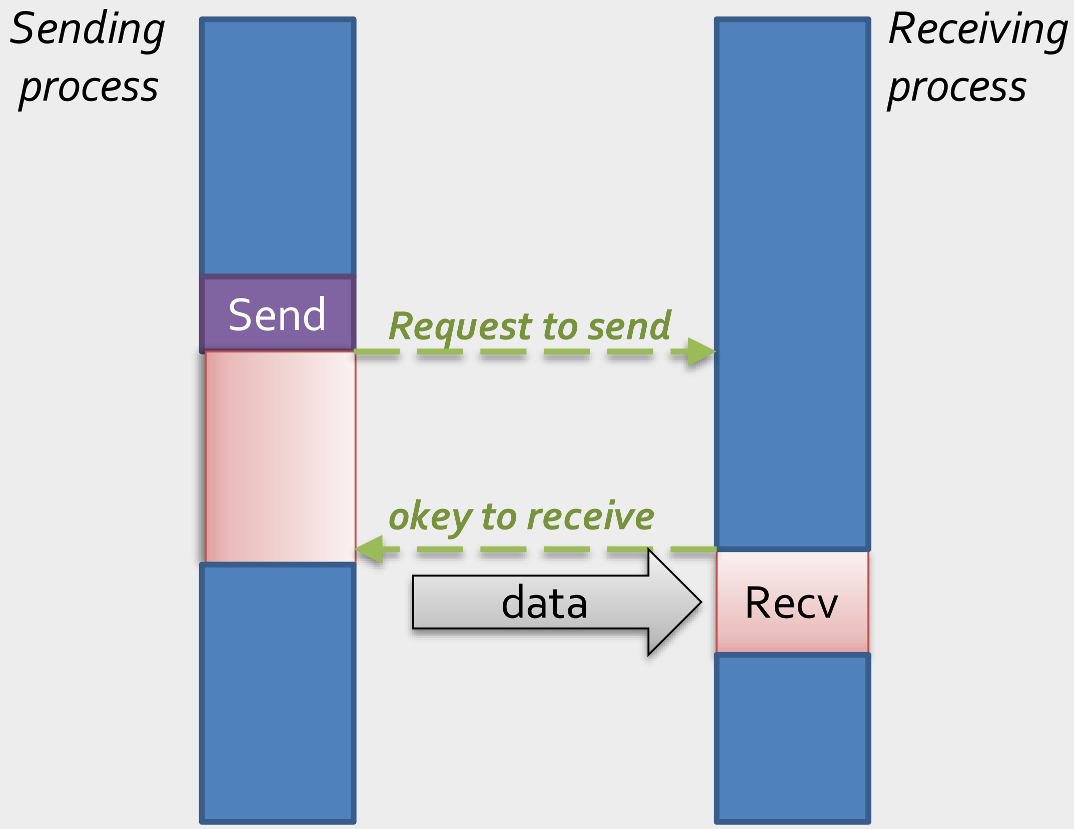
With ***blocking*** operation the process blocks until some *condition* is achieved.
This ***condition*** depends on type of blocking operation and the library implementation.

So far we have discussed ***blocking*** communication:

- **MPI_Send**: does not complete until buffer is empty (or available for reuse).
- **MPI_Recv**: receive a message and block until the requested data is available in the application buffer in the receiving task
- **MPI_Probe**: performs a blocking test for a message.

Blocking operations

- **MPI_Ssend – *synchronous blocking send***: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message



Blocking operations

```
int main(int argc, char *argv[]) {
    int rank, size, I, tag = 0;
    int buffer[10];
    MPI_Status status;
    /* MPI Initialization*/
    if (rank == 0) {
        for (i=0; i<10; i++)
            buffer[i] = i;
        MPI_Ssend(buffer, 10, MPI_INT, 1, tag, MPI_COMM_WORLD);
    }
    if (rank == 1) {
        for (i=0; i<10; i++)
            buffer[i] = -1;
        MPI_Recv(buffer, 10, MPI_INT, 0, tag, ..., &status);
        for (i=0; i<10; i++) {
            if (buffer[i] != i)
                printf("Error: buffer[%d] = %d but is expected to
                            be %d\n", i, buffer[i], i);
        }
        fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```

Blocking operations

- **MPI_Bsend – *buffered blocking send***: permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered.
- Routine returns after the data has been copied from application buffer space to the allocated send buffer.
- Must be used with the MPI_Buffer_attach routine:

```
int MPI_Buffer_attach(void *buf, int size),
➢ buf - initial buffer address
➢ size - buffer size, in bytes
```

- After message has sent the buffer should be removed:

```
int MPI_Buffer_detach(void *buf, int *size)
```

Blocking operations

```
int main(int argc,char *argv[]) {
   int *buffer;
   int myrank, buffsize = 1, TAG = 0;
   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
   if (myrank == 0){
      buffer = (int *) malloc(buffsize + MPI_BSEND_OVERHEAD);
      MPI_Buffer_attach(buffer, buffsize + MPI_BSEND_OVERHEAD);
      buffer = (int *) 10;
      MPI_Bsend(&buffer, buffsize, MPI_INT, 1, TAG,  ...);
      MPI_Buffer_detach(&buffer, &buffsize);
   }
   else {
      MPI_Recv(&buffer, buffsize, MPI_INT, 0, TAG,..., &status);
      printf("received: %i\n", buffer);
   }
   MPI_Finalize();
   return 0;
}
```

Let’s consider

```
/*   ...   */

int j, i= (rank+1)*5;
 if (rank==0)
 {
  MPI_Recv(&j,1,MPI_INT,1,message2,MPI_COMM_WORLD,&status);
  MPI_Send(&i,1,MPI_INT,1,message1,MPI_COMM_WORLD);
}
 if (rank==1)
 {
  MPI_Recv(&j,1,MPI_INT,0,message1,MPI_COMM_WORLD,&status);
  MPI_Send(&i,1,MPI_INT,0,message2,MPI_COMM_WORLD);
 }

/*   ...   */
```

What’s the result?

Deadlocks

```
Process 0:
RECEIVE(1)
SEND(1)
```

```
Process 1:
RECEIVE(0)
SEBD(0)
```

```
Process 0:
SEND(1)
RECEIVE(1)
```

```
Process 1:
SEND(0)
RECEIVE(0)
```

Avoiding Deadlocks

1. Order the operations more carefully:

Process 0:	Process 1:
MPI_Recv(1) **MPI_Send(1)**	**MPI_Send(0)** **MPI_Recv(0)**

2. Use non-blocking operations:

Process 0:	Process 1:
MPI_Irecv(1) **MPI_Send (1)**	**MPI_Send(0)** **MPI_Recv(0)**

3. Supply receive buffer at same time as send, with MPI_Sendrecv:

Process 0:	Process 1:
MPI_Sendrecv(1)	**MPI_Sendrecv(0)**

Non-Blocking Operations

- Return straight away and allow the sub-program to continue to perform other work
- Avoids many common dead-lock situations
- You can mix non-blocking and blocking routines

```
int MPI_Isend( buf, count, type, dest, tag, comm, request)
```

- **buf** – send buffer that must not be written to until one has checked that the operation is over
- **request** – a handle that is used when checking if the operation has finished (MPI_Request in C)

```
int MPI_Irecv( buf, count, type, dest, tag, comm, request)
```

- **buf** – receive buffer guaranteed to contain the data only after one has checked that the operation is over
- **request** – a handle that is used when checking if the operation has finished

Non-Blocking Operations

- Other non-blocking operations:

```
int MPI_Issend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request)


int MPI_Ibsend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request)


int MPI_Irsend(void *buf, int count, MPI_Datatype type,
      int dest, int tag, MPI_Comm comm, MPI_Request *request)


int MPI_Iprobe(int source, int msgtag, MPI_Comm comm,
                                 int *flag, MPI_Status *status);
```

Non-Blocking Operations

- Non-blocking operations are generally accompanied by a check-status operation, which indicates whether the semantics of a previously initiated transfer may be violated or not:

```
int MPI_Test(request, flag,     status)

    ➢ request - communication request,
    ➢ flag    - true if operation completed,
    ➢ status  - status for the completed operations
```

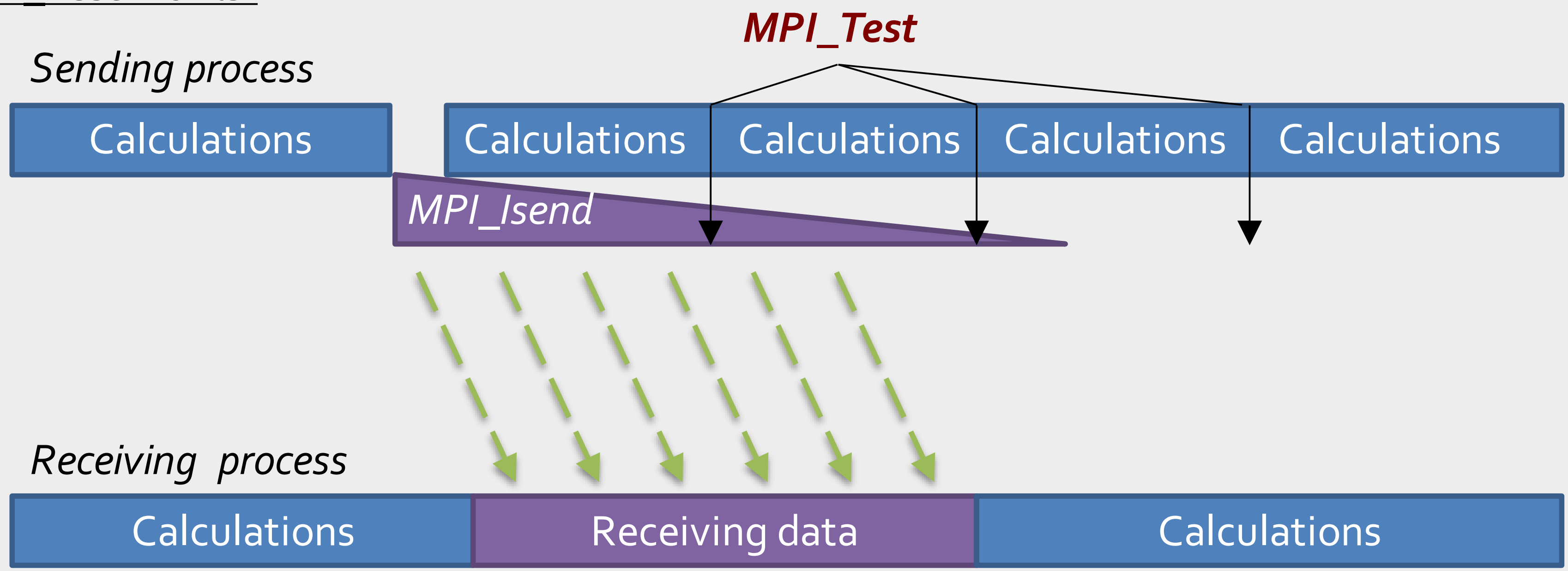
- A call to MPI_Test is non-blocking. It allows one to schedule alternative activities while periodically checking for completion

- **Additional completion Test-operations**:

```
MPI_Testall  - Tests  for  the completion of all previously
        initiated communications
MPI_Testany  - Tests for completion of any previously initiated
               communication
MPI_Testsome - Tests for one or more given communications to
               complete
```

Non-Blocking Operations

How MPI_Test works:



MPI_Test commonly using:

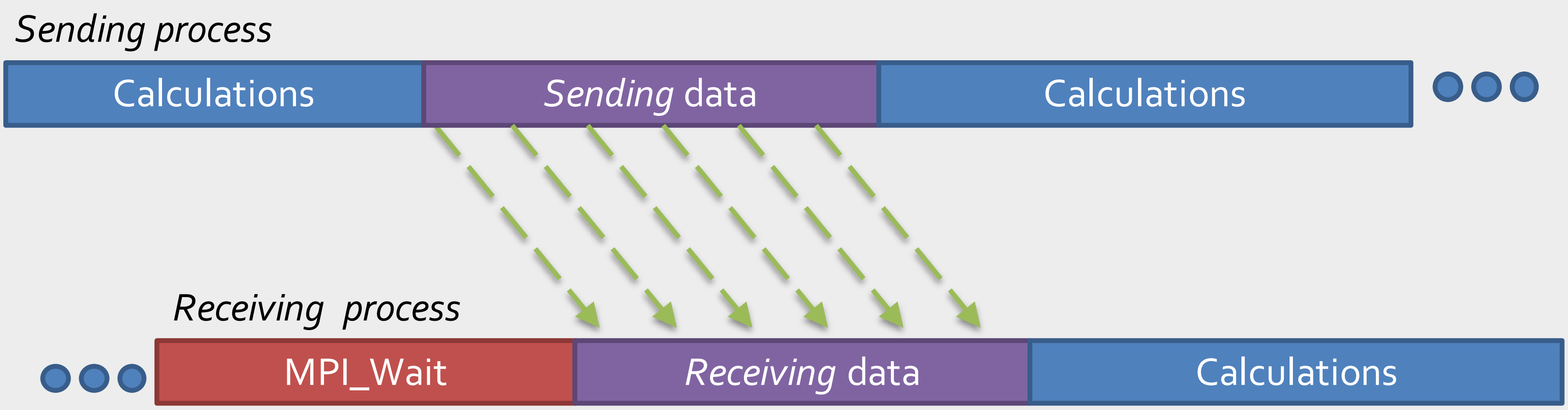
```
MPI_Isend(buf,count,type,dest,tag,comm,&request);
…
do {
  …
  MPI_Test(&request,&flag,&status);
} while ( !flag );
```

Non-Blocking Operations

- Blocking check-status operation:

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

How MPI_Wait works:



- A call to MPI_WAIT returns when the operation identified by request is comlete
- After function's returns request is set to MPI_REQUEST_NULL

Non-Blocking Operations

- **Additional completion Wait-operations**:

```
MPI_Waitall  - returns when all operations identified by the
               array of requests are complete

MPI_Waitany  - returns when one operation identified by the array
               of requests is complete

MPI_Waitsome - returns when one or more operation identified by
               the array of requests is complete
```

Non-Blocking Operations

```
int main(int argc, char** argv){
  int numtasks, rank, next, prev, buf[2], tag_1 = 1, tag_2 = 2;
  MPI_Request reqs[4];
  MPI_Status stats[4];
  /* MPI Initialization */
  prev = rank - 1;
  next = rank + 1;
  if (rank == 0)  prev = numtasks - 1;
  if (rank == (numtasks - 1))  next = 0;

  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag_1, comm, &reqs[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag_2, comm, &reqs[1]);

  MPI_Isend(&rank, 1, MPI_INT, prev, tag_2, comm, &reqs[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag_1, comm, &reqs[3]);

  MPI_Waitall(4, reqs, stats);
  printf("Node %d: all ok!\n", rank);
  MPI_Finalize();
}
```

Combined Send/Receive

- Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message

```
int MPI_Sendrecv(
  void *sbuf, int scount, MPI_Datatype stype, int dest, int stag,
  void *rbuf, int rcount, MPI_Datatype rtype, int source,int rtag
  MPI_Comm comm, MPI_Status *status),

- sbuf, scount, stype, dest,   stag – characteristics of sending
                                      message

- rbuf, rcount, rtype, source, rtag – characteristics of receiving
                                      message
- comm – communicator,
- status – status object.
```

Combined Send/Receive

```
int main(int argc, char** argv){
  int numtasks, rank, next, prev, buf[2], tag_1 = 1, tag_2 = 2;
  MPI_Request rbuf [2];
  MPI_Status status_1, status_2;
/* MPI Initialization*/

  prev = rank - 1;
  next = rank + 1;
  if (rank == 0) prev = numtasks - 1;
  if (rank == (numtasks - 1)) next = 0;

  MPI_Sendrecv(&sbuf[0], 1, MPI_FLOAT, prev, tag_2, &rbuf[0], 1,
               MPI_FLOAT, next, tag2, MPI_COMM_WORLD, &status_1);

  MPI_Sendrecv(&sbuf[1], 1, MPI_FLOAT, next, tag_1, &rbuf[1], 1,
               MPI_FLOAT, prev, tag1, MPI_COMM_WORLD, &status_2);
  printf("Node %d: all ok!\n", rank);
  MPI_Finalize();
}
```

A simple calculation problem

x: | x_1 | x_2 | x_3 | x_4 | ••• | x_i | ••• | x_{n-1} | x_n |

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

How can it be solved with MPI?

(considering that we have a distributed system with N processors/nodes/computers)

To solve this problem with MPI it is required to:

- pass whole vector to processes
- pick out a part of the data (depending on process Rank) in each process
- perform a summation of the data block in each process
- collect values of calculated partial sums in one of the processes
- add the values of partial sums to get the total result

Some simple calculation problem

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

The solution with MPI:

1) pass whole vector to processes:

```
MPI_Comm_size (MPI_COMM_WORLD, &ProcNum);
if ( ProcRank == 0 )
     for (i = 1; i < ProcNum; i++)
           MPI_Send(&x, n, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
else
           MPI_Recv(&x, n, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &Status);
```

2) pick out a part of the data (depending on process Rank)
3) perform a summation of the data block in each process

```
count = N/ProcNum;
from_idx = count * ProcRank;
to_idx = count * (ProcRank + 1);
if (ProcRank == ProcNum-1)
          to_idx  = N;
for (i = from_idx; i<to_idx; i++)
         ProcSum  = ProcSum + x[i];
```

For instance

N=12 *ProcNum=4* ⇒ *count=3*

ProcRank	from_idx	to_idx
0	0	3
1	3	6
2	6	9
3	9	12

Some simple calculation problem

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

The solution with MPI :

4) collect values of calculated partial sums in one of the processes

5) add the values of partial sums to get the total result

```
if ( ProcRank == 0 ) {  // One process collects partial sums
      TotalSum = ProcSum;
      for ( i = 1; i < ProcNum; i++ ) {
              MPI_Recv(&ProcSum, 1, MPI_DOUBLE, i, MPI_ANY_TAG,

         MPI_COMM_WORLD, &Status);
              TotalSum = TotalSum + ProcSum;
      }
}
else    // Each process sends its partial sum
       MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, some_tag, MPI_COMM_WORLD);
```

Some simple calculation problem

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

<u>The solution with MPI :</u>

4) collect values of calculated partial sums in one of the processes

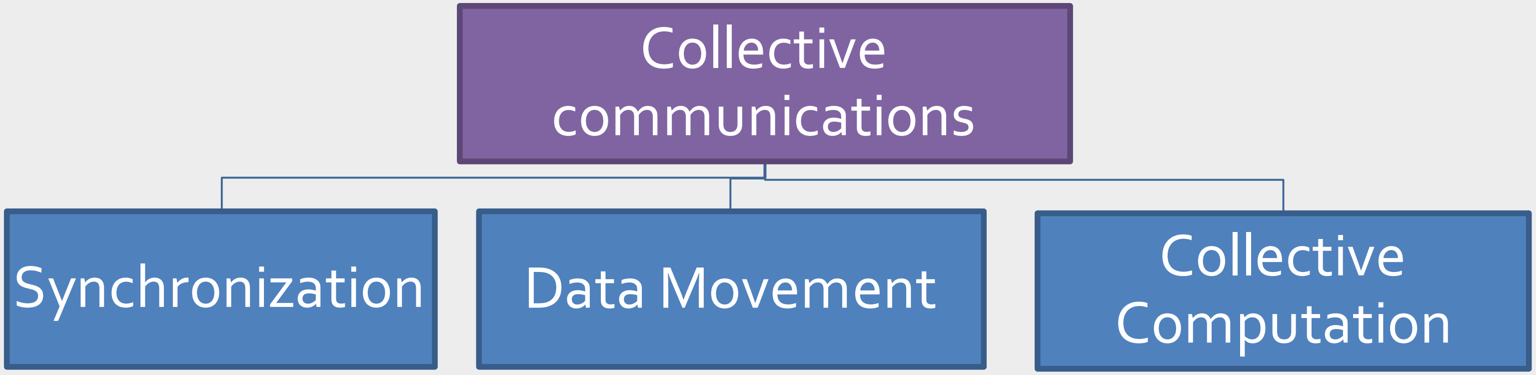
5) add the values of partial sums to get the total result

```
if ( ProcRank == 0 ) {  // One process collects partial sums
      TotalSum = ProcSum;
      for ( i = 1; i < ProcNum; i++ ) {
            MPI_Recv(&ProcSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                                       MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
            TotalSum = TotalSum + ProcSum;
      }
}
else    // Each process sends its partial sum
      MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, some_tag, MPI_COMM_WORLD);
```

Are there any disadvantages?

Collective Communications

- Collective communication routines must involve **all** processes within the scope of a *communicator*.
- Unexpected behavior, including program failure, might occur if even one task in the communicator doesn't participate.



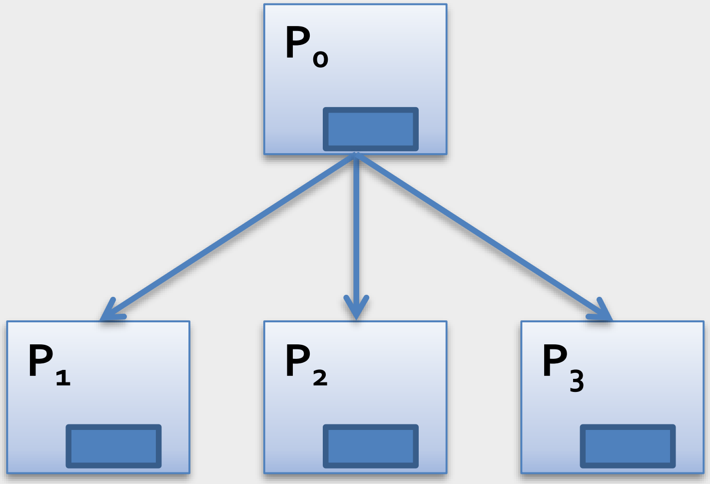
- Can only be used with MPI predefined datatypes.
- With MPI-3, collective operations can be blocking or non-blocking.
- Collective communication routines do not take message tag arguments.

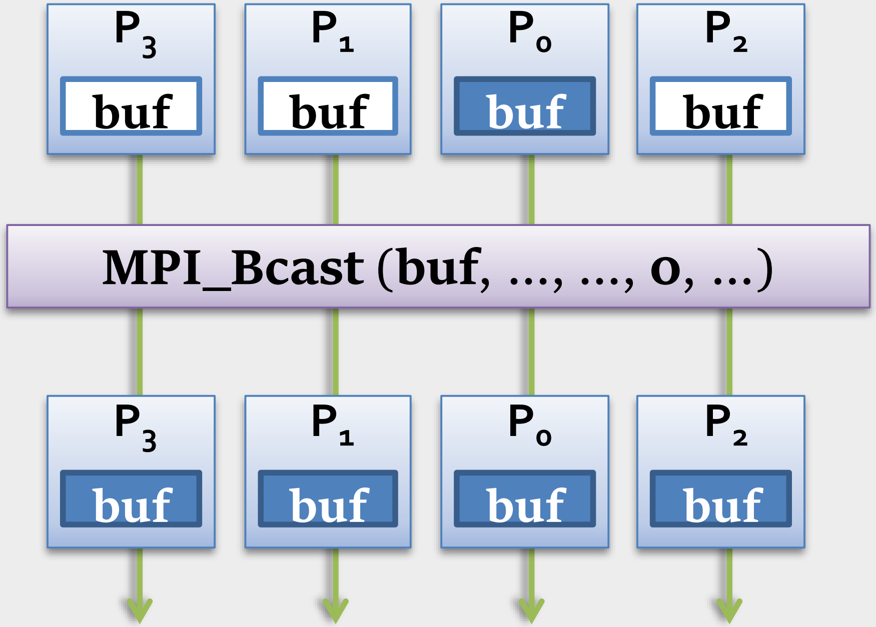
Collective Communications

- Data movement operation 'broadcast':

```
int MPI_Bcast (&buffer, count, datatype, root, comm);
```

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





Collective Communications

Calculate the sum of vector's elements:

$$S = \sum_{i=1}^{n} x_i$$

The solution with MPI:

1) pass whole vector to processes:

```
MPI_Comm_size (MPI_COMM_WORLD, &ProcNum);
if ( ProcRank == 0 )
      for (i = 1; i < ProcNum; i++)
            MPI_Send(&x, N, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
else
            MPI_Recv(&x, N, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD,
&Status);
```

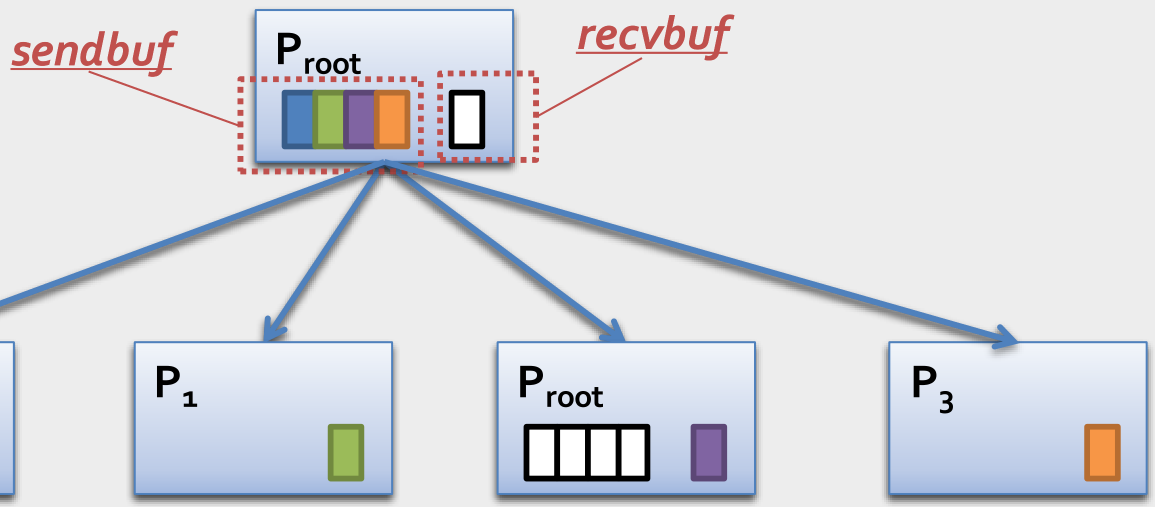
```
MPI_Comm_size (MPI_COMM_WORLD, &ProcNum);

MPI_Bcast (&x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Does each process need a whole vector?

Collective Communications

- Data movement operation 'scatter' distributes distinct messages from a 'root' to each process in the communicator.

```
int MPI_Scatter (sendbuf, sendcnt, sendtype,
                         &recvbuf, recvcnt, recvtype, root, comm)

sendbuf: address of send buffer     recvbuf: address of receive buffer
sendcnt: number of elements sent    recvcnt: number of elements in
receive to each process                                    buffer
```



- MPI_Scatterv is used with variable data size

Collective Communications

Calculate the sum of vector's elements:

$$S = \sum_{i=1}^{n} x_i$$

The solution with MPI:

1) pass whole vector to processes
2) pick out a part of the data (depending on process Rank)

1) split the data into equal blocks
2) pass these blocks to processes

```
float vec_X [N];
Block_SIZE= N / ProcNum;
float vec_Part [Block_SIZE];
MPI_Scatter (vec_X, Block_SIZE, MPI_FLOAT, vec_Part, Block_SIZE,
                                    MPI_FLOAT, 0, MPI_COMM_WORLD);
```

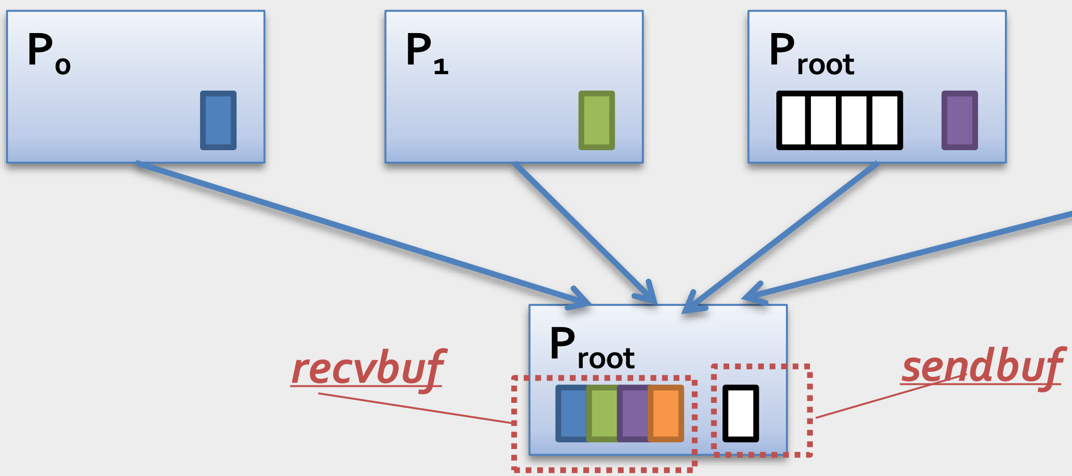
3) perform a summation of the data block in each process

```
for (i = 0 ; i < Block_SIZE; i++ )
        ProcSum  = ProcSum + vec_Part[i];
```

Collective Communications

- Data movement operation 'gather' gathers distinct messages from each process in the communicator to a single destination process.

```
int MPI_Gather (&sendbuf, sendcnt, sendtype,
                       &recvbuf, recvcnt, recvtype, root, comm)

sendbuf: address of send buffer        recvbuf: address of receive buffer
sendcnt: number of elements in         recvcnt: number of elements from
each send buffer                                process
```



- MPI_Gatherv is used with variable data size

Collective Communications

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

<u>The solution with MPI :</u>

4) collect values of calculated partial sums in one of the processes
5) add the values of partial sums to get the total result

```
if ( ProcRank == 0 ) {       // One process collects partial sums
     TotalSum = ProcSum;
     for ( i = 1; i < ProcNum; i++ ) {
           MPI_Recv(&ProcSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                    MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
           TotalSum = TotalSum + ProcSum;
     }
}
else   // Each process sends its partial sum
      MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, some_tag,
                                        MPI_COMM_WORLD);
```

Collective Communications

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

<u>The solution with MPI :</u>

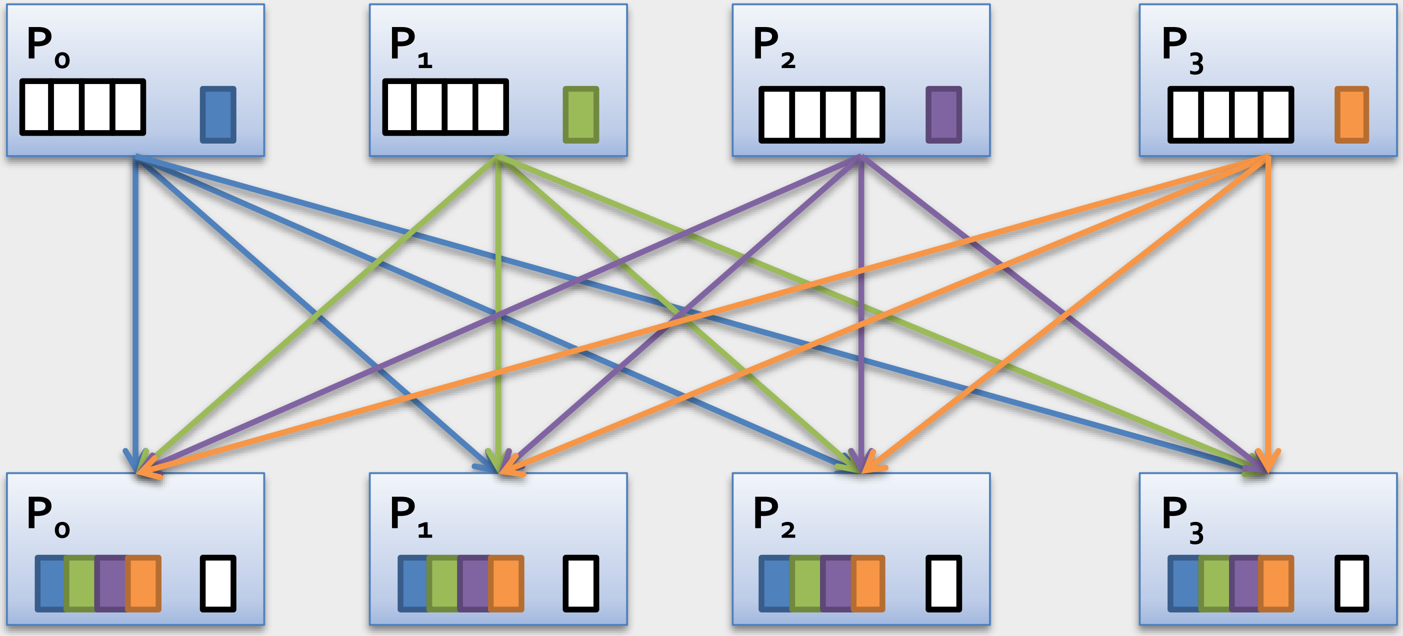
4) collect values of calculated partial sums in one of the processes
5) add the values of partial sums to get the total result

```
…
float *part_Sum
if ( ProcRank == 0 )
     part_Sum = new float[ProcNum]
MPI_Gather (ProcSum, 1, MPI_FLOAT,
                     part_Sum, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

if ( ProcRank == 0 ) {
   TotalSum = 0;
     for ( i = 1; i < ProcNum; i++ )
       TotalSum += part_Sum[i];
}
```

Collective Communications

- Data movement operation 'all-gather': Concatenation of data to all process in a group.

```
int MPI_Allgather (&sendbuf, sendcnt, sendtype, &recvbuf,
                                  recvcnt, recvtype, comm)
```

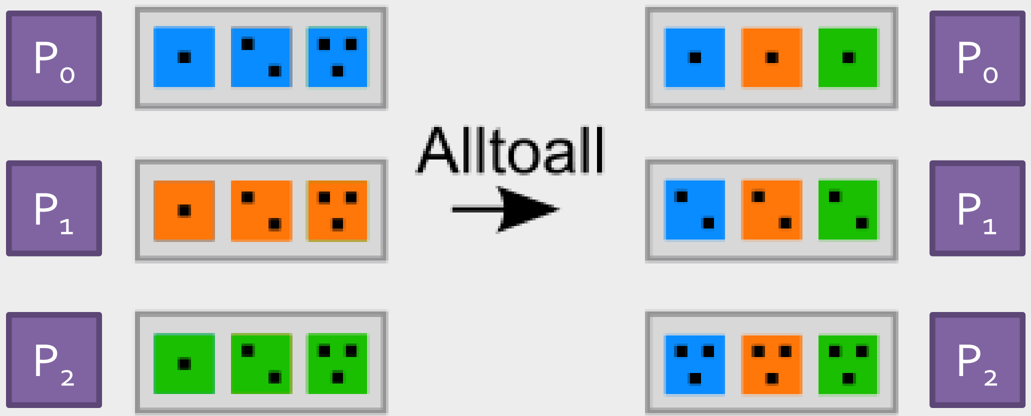


- MPI_Allgatherv is used with variable data size

Collective Communications

- Data movement operation 'all-to-all' sends data from all to all processes

```
int MPI_Alltoall (&sendbuf, sendcnt, sendtype, &recvbuf,
                                    recvcnt, recvtype, comm)
```

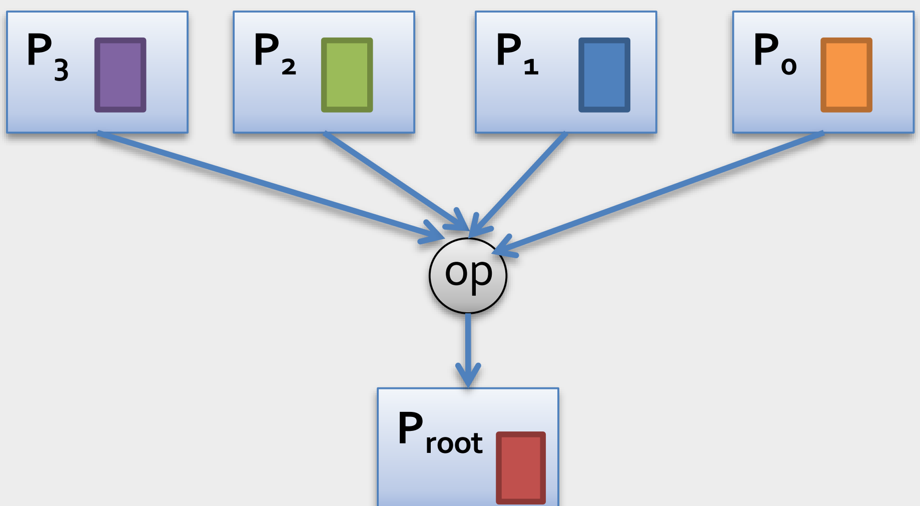


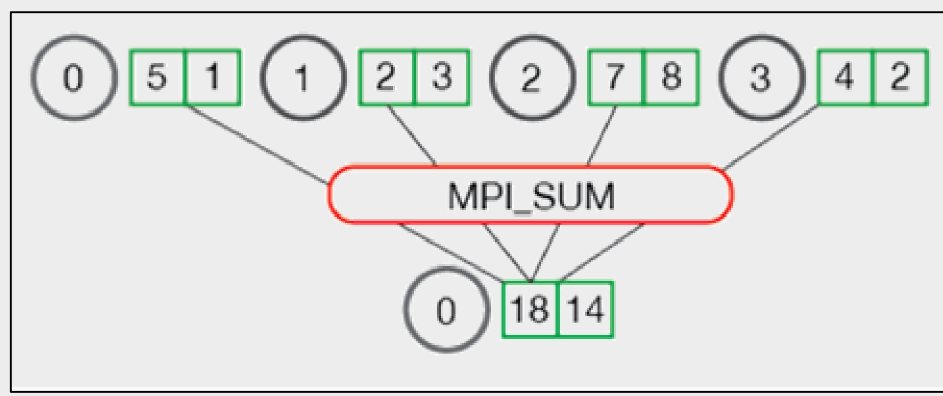
- MPI_Alltoallv is used with variable data size

Collective Communications

- Collective computation operation 'reduction'
  applies a reduction operation on all process in the group and places the result in one process

```
int MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, root, comm)

sendbuf:address of send buffer              recvbuf:address of receive buf.
count: number of elements in send buffer    op: reduce operation
datatype: data type of send buffer          root: rank of root process
```





Collective Communications

The predefined MPI reduction operations	
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

- Users can also define their own reduction functions by using the MPI_Op_create routine

Collective Communications

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

<u>The solution with MPI :</u>

4) collect values of calculated partial sums in one of the processes
5) add the values of partial sums to get the total result

```
…
float *part_Sum
if ( ProcRank == 0 )
     part_Sum = new float[ProcNum]
MPI_Gather (ProcSum, 1, MPI_FLOAT,
                     part_Sum, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

if ( ProcRank == 0 ) {
   TotalSum = 0;
     for ( i = 1; i < ProcNum; i++ )
       TotalSum += part_Sum[i];
}
```

Collective Communications

Calculate the sum of vector's elements: $S = \sum_{i=1}^{n} x_i$

The solution with MPI :

4) collect values of calculated partial sums in one of the processes

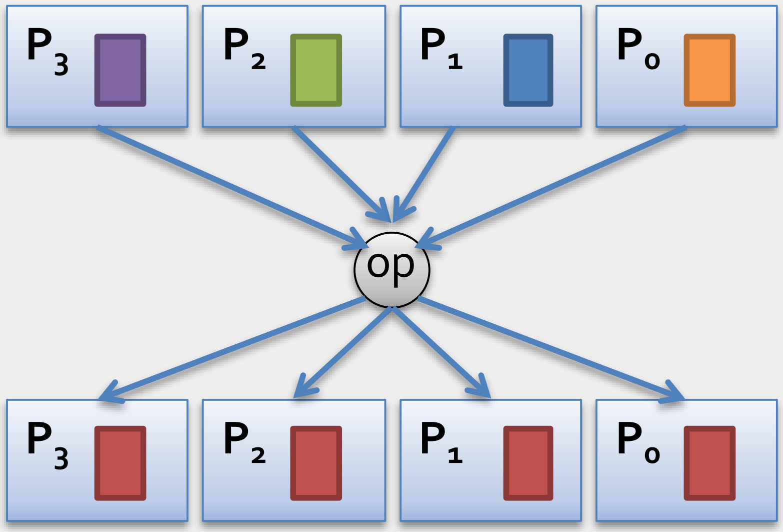
5) add the values of partial sums to get the total result

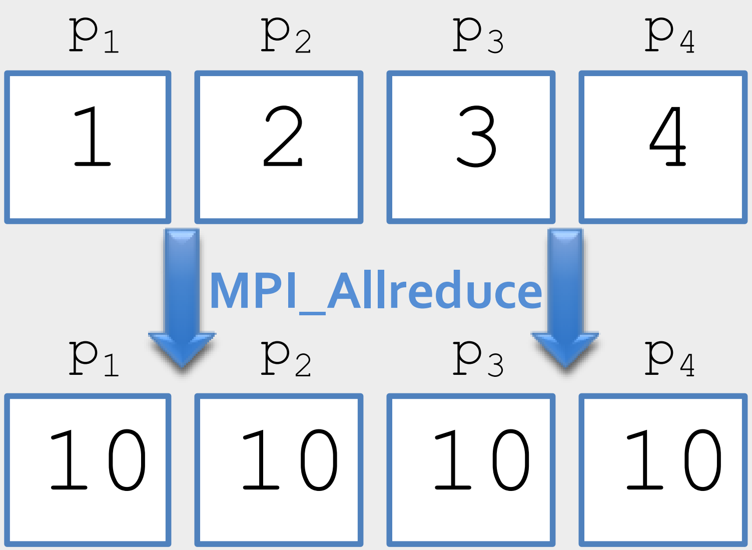
```
...
MPI_Reduce (&ProcSum,&TotalSum,1,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD)
```

Collective Communications

- Collective computation operation + data movement 'all-reduce' combines values from all processes and distribute the result back to all processes

```
int MPI_AllReduce (&sendbuf, &recvbuf, count, datatype, op, comm)
```



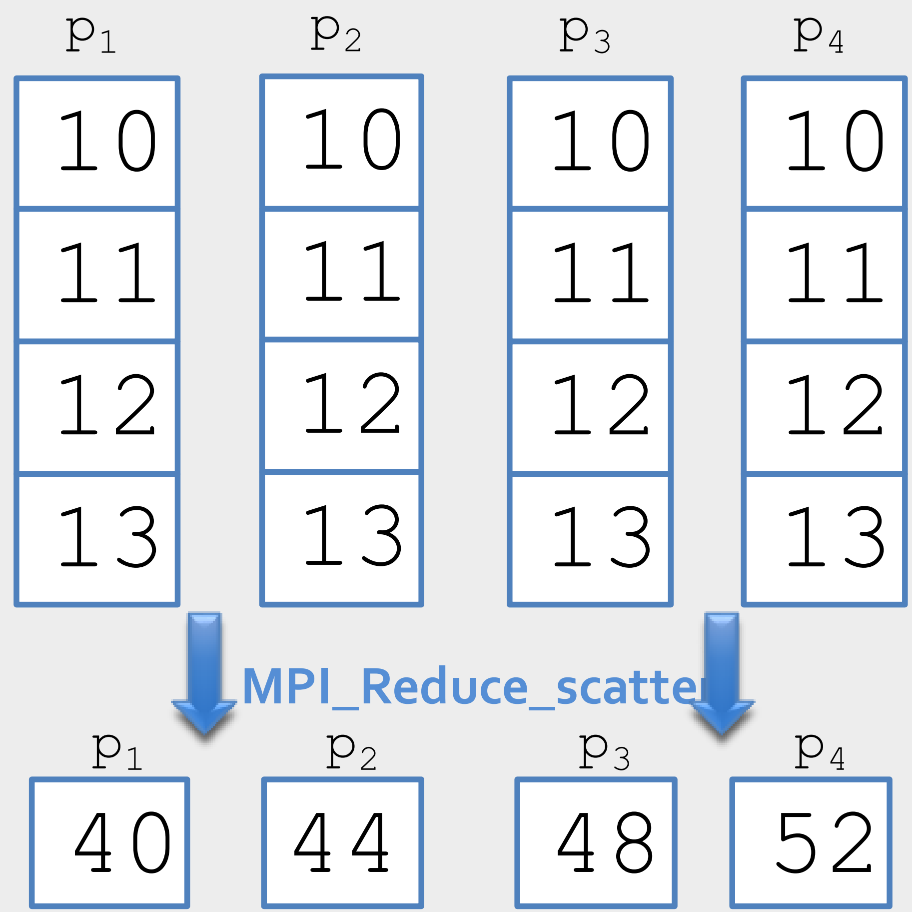


Collective Communications

- Collective computation operation + data movement ‘reduce-scatter’
  - First it does an element-wise reduction on a vector across all processors.
  - Next, the result vector is split into disjoint segments and distributed across the processors

int **MPI_Reduce_scatter** (&sendbuf,
&recvbuf, recvcounts,
datatype, op, comm)

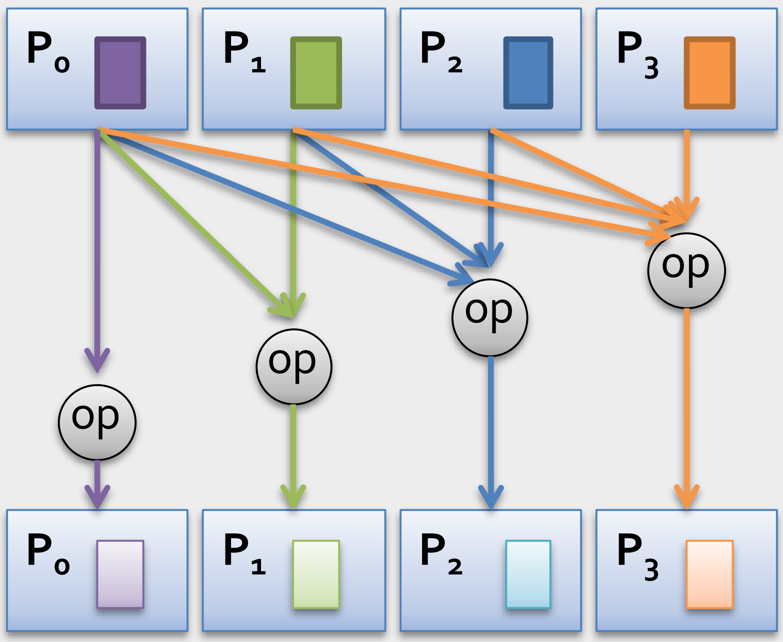
sendbuf: address of send buffer
recvbuf: address of receive buffer
recvcounts: integer array specifying the number of elements in result distributed to each process
datatype: data type of send buffer
op: reduce operation

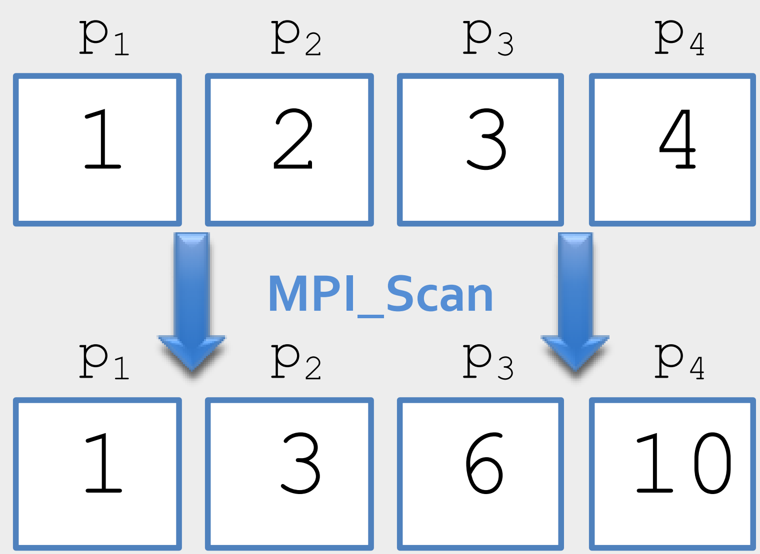


Collective Communications

- Collective computation operation ‘scan’ computes the scan (***partial reductions***) of data on a collection of processes

```
int MPI_Scan (&sendbuf, &recvbuf, count, datatype, op, comm)
```



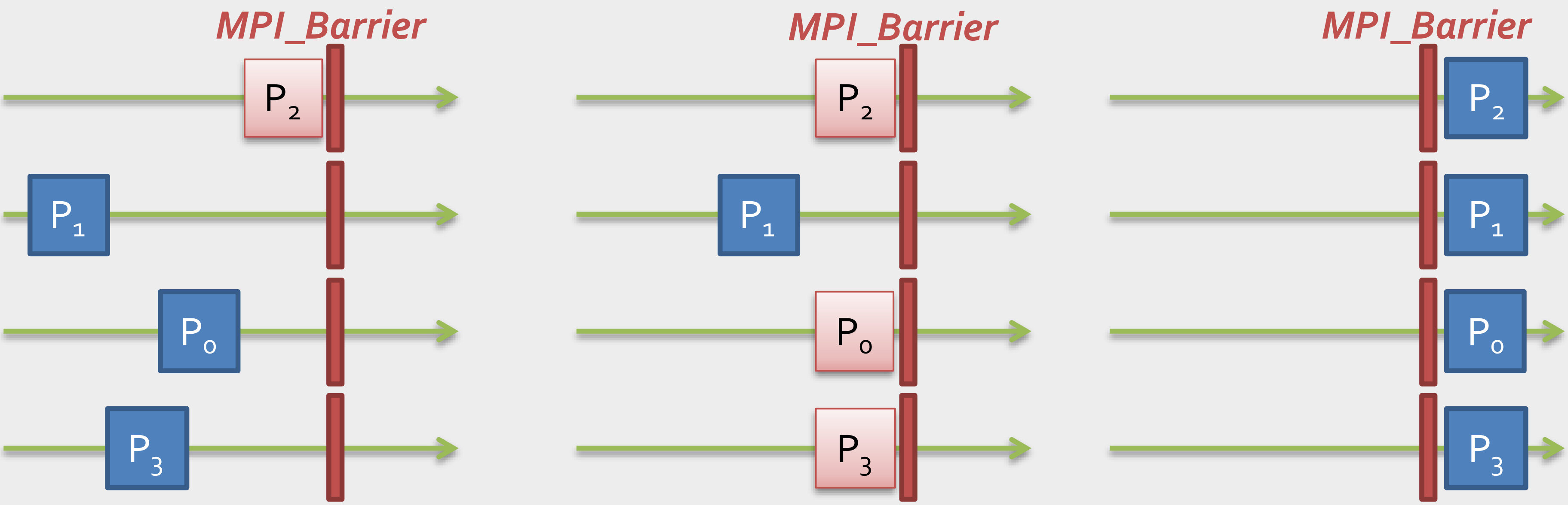


Collective Communications

- Synchronization:

```
int MPI_Barrier (comm);
```

- Creates a barrier synchronization in a communicator
- Each process, when reaching the MPI_Barrier call, is blocked until all processors in the communicator reach the same MPI_Barrier call

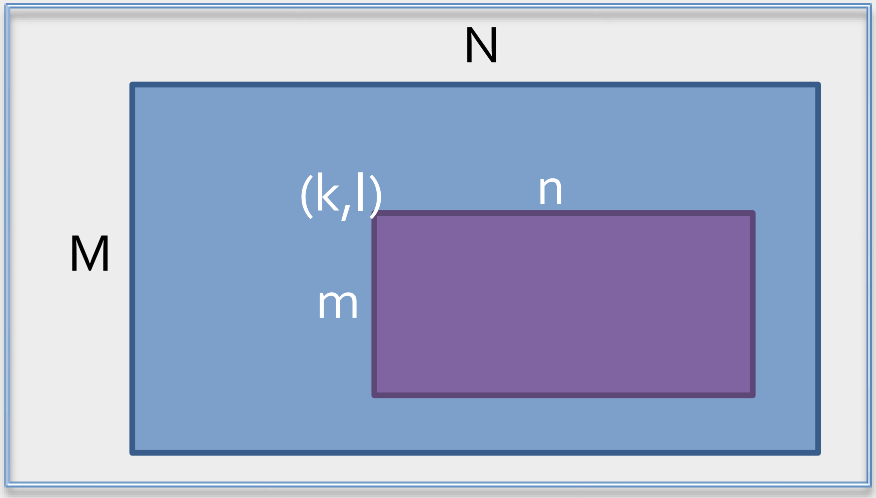


Transferring heterogeneous data

- Up to here, we consider the MPI-message as a buffer containing a sequence of identical basic data types.
- But often it needs to transfer efficiently heterogeneous and noncontiguous data

For instance (1):

- We need to send noncontiguous data (e.g., a sub-block of a matrix)



For instance (2):

- We need to pass messages that contain values with different datatypes

```
int    i;
double d;
char   c;
```

Transferring heterogeneous data

- ***1st Way:*** Send each element separately (one after another):

```
MPI_Send(i, 1, MPI_INT,    tgRank, msgTag_1, comm);
MPI_Send(d, 1, MPI_DOUBLE, tgRank, msgTag_2, comm);
MPI_Send(c, 1, MPI_CHAR,   tgRank, msgTag_3, comm);
```

- *And receive each element one after another*

- *Advantages*:
  - Simple to implement

- *Disadvantages*:
  - Sending a large number of messages
  - Losing performance
  - Deadlock could arise

Transferring heterogeneous data

- ***2nd Way:*** Using programing language features to buffer values before passing message

```
p = &buffer;
for (i=0; i<n; ++i)
  for (j=0; j<m; ++j)
    *(p++) = a[i + k][j + l];

MPI_Send(p, n*m, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
```

- *Disadvantages*:
  - Memory and copying data Overhead
  - Only one datatype is sent
  - Makes code less readable

Transferring heterogeneous data. Pack/Unpack

- <u>*3rd Way:*</u> pack noncontiguous data into a contiguous buffer at the sender site and unpack it at the receiver site

- Packing the message in the send buffer

MPI_Pack (inbuf, incount, datatype, outbuf, buflen, pos, comm)

inbuf: input buffer start
incount: number of in-data items
datatype: datatype of each input data item
outbuf: output buffer start
buflen: output buffer size
pos: current position in buffer

- <u>Example of sender site:</u>

```
int bufPos = 0;
MPI_Pack(i, 1, MPI_INT,   buf, bufLen, &bufPos, comm);
MPI_Pack(d, 1, MPI_DOUBLE,buf, bufLen, &bufPos, comm);
MPI_Pack(c, 1, MPI_CHAR,  buf, bufLen, &bufPos, comm);

MPI_Send( buf, bufPos, MPI_PACKED, tgRank, msgTag, comm);
```

Pack/Unpack Data

- Unpacking a message into the receive buffer:

MPI_Unpack(inbuf, insize, pos, outbuf, outcount, datatype, comm)

inbuf: input buffer start
insize:size of input buffer
pos: current position in buffer
outbuf: output buffer start
outcount: number of items to be unpacked
datatype: datatype of each output data item

- Example of receiver site

```
int bufPos = 0;
MPI_Recv(buf, bufSize, MPI_PACKED, srcRank, mTag, comm, &status);

MPI_Unpack(buf, bufLen, &bufPos, &i, 1, MPI_INT,    comm);
MPI_Unpack(buf, bufLen, &bufPos, &d, 1, MPI_DOUBLE, comm);
MPI_Unpack(buf, bufLen, &bufPos, &c, 1, MPI_CHAR,   comm);
```

Pack/Unpack Data

- The following call allows the user to find out how much space is needed to pack a message and, thus, manage space allocation for buffers.

```
int MPI_Pack_size(incount, datatype, comm, size)

incount:   count argument to packing call
datatype:  datatype argument to packing call
size:      upper bound on size of packed message, in bytes
```

Sample:

```
int bufSize = 0;
    void *tempBuf;
    MPI_Pack_size( 1, MPI_INT,    MPI_COMM_WORLD, &bufSize );
    MPI_Pack_size( 1, MPI_DOUBLE, MPI_COMM_WORLD, &bufSize );
    MPI_Pack_size( 4, MPI_CHAR,   MPI_COMM_WORLD, &bufSize );

tempBuf = malloc( bufSize );
```

MPI Derived Data Types

- More general communication buffers are specified by replacing the basic data types that have been used so far with ***derived data types*** that are constructed from basic data types using the constructors
- These methods of constructing derived data types can be applied recursively
- A derived data type is an opaque object that specifies two things:
  - a sequence of basic data types
  - a sequence of integer (byte) displacements

↓

TypeMap = $\{(type_0, disp_0), (type_1, disp_1), \ldots, (type_{n-1}, disp_{n-1})\}$

- Using the derived type in the message passing functions can be regarded as the template that helps interpret the data

MPI Derived Data Types

General usage of MPI derived data types:

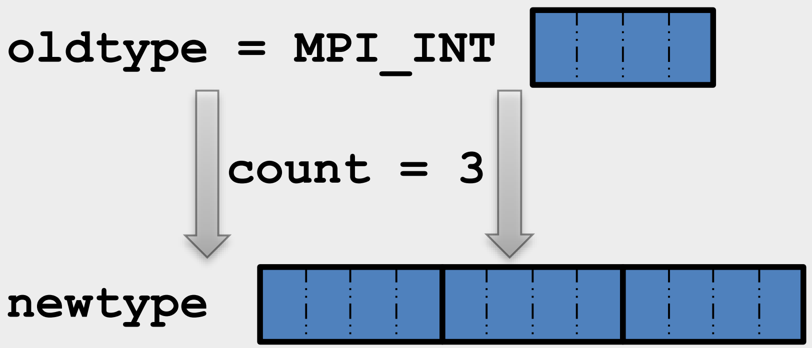
- Construction derived data type
  - *MPI_Type_contiguous*
  - *MPI_Type_vector*
  - *MPI_Type_indexed*
  - *MPI_Type_struct*
- Commit new datatype to the system
  - *MPI_Type_commit*
- Send and receive messages with new type
- Free the datatype
  - *MPI_Type_free*

MPI Derived Data Types

- ***Contiguous*** constructor is the simplest constructor.
- Produces a new data type by making count copies of an existing data type.

int **MPI_Type_contiguous**(count, oldtype, &newtype);

- **count** – replication count
- **oldtype** – old datatype
- **newtype** – new datatype.



MPI Derived Data Types

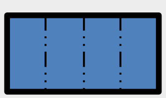
```
int main() {
   int rank;
   struct { int x;
        int y;
        int z;
   } point;
   MPI_Datatype ptype;
   MPI_Init(&argc,&argv); MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   MPI_Type_contiguous(3,MPI_INT,&ptype);
   MPI_Type_commit(&ptype);
   if (rank==3){
         point.x = 45; point.y = 36; point.z = 0;
         MPI_Send(&point,1,ptype,1,mTag,MPI_COMM_WORLD);
   }
   else if (rank==1) {
         MPI_Recv(&point,1,ptype,3,mTag,MPI_COMM_WORLD,&status);
         printf("Proc №%d received point with coords (%d;%d;%d)

       \n",rank,point.x,point.y,point.z);
   }
MPI_Finalize();
}
```

MPI Derived Data Types

- ***Vector*** constructor is a more general constructor that allows replication of a data type into locations that consist of equally spaced blocks.

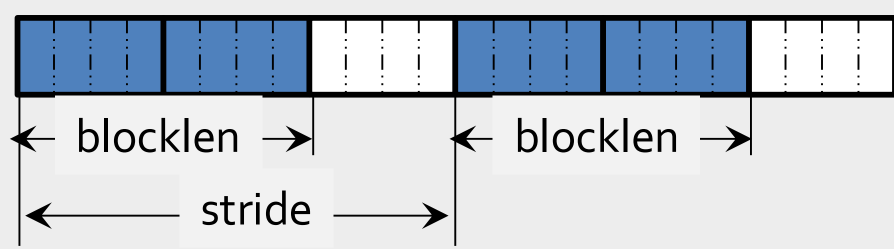
```
MPI_Type_vector (count, blocklen, stride, oldtype, *newtype )

- count    - number of blocks,
- blocklen - number of elements in each block,
- stride   - number of elements between start of each block
- oldtype  - old datatype,
- newtype  - new datatype.
```

oldtype = MPI_INT



count = 2,
blocklen = 2,
stride = 3

newtype



MPI Derived Data Types

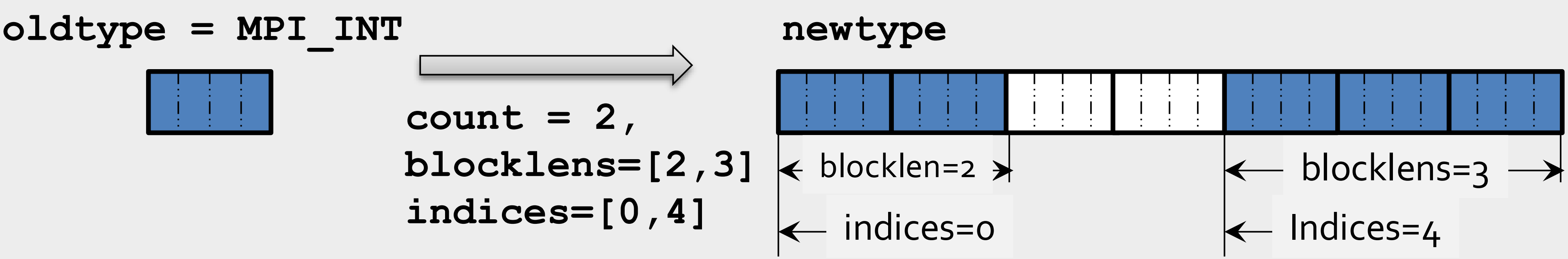
```
int main(int argc, char *argv[]) {
    int rank,i,j;
    double x[4][8];
    MPI_Datatype coltype;
...
    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);
    if(rank==3){
        for(i=0;i<4;++i)
            for(j=0;j<8;++j)
                x[i][j]=pow(10.0,i+1)+j;
        MPI_Send(&x[0][7],1,coltype,1,mTag,MPI_COMM_WORLD);
    }
    else if(rank==1) {
        MPI_Recv(&x[0][2],1,coltype,3,mTag,MPI_COMM_WORLD,&status);
        for(i=0;i<4;++i)
            printf("Proc №%d: my x[%d][2]=%1f\n",rank,i,x[i][2]);
    }
MPI_Finalize();
}
```

MPI Derived Data Types

- ***Indexed*** constructor allows replication of an old datatype into a sequence of blocks, where each block can contain a different number of copies and have a different displacement.

int **MPI_Type_indexed** (count, blocklens[], indices[], oldtype, *newtype)

count: number of blocks,
blocklens: number of elements in each block,
indices: displacement of each block (in multiples of old_type),
oldtype: old datatype,
newtype: new datatype.



MPI Derived Data Types

```
#define  SIZE  100
float a[ SIZE ][ SIZE ];
int pos[ SIZE ]
int len[ SIZE ];
MPI_Datatype upper;
...
for( i=0; i<SIZE; i++ ) {    /*   xxxxxx   */
        pos[i] = SIZE*i + i;     /*   .xxxxx   */
        len[i] = SIZE - i;          /*   ..xxxx   */
}                                   /*   ...xxx   */

MPI_Type_indexed(
    SIZE,                          /* number of arrays*/
    len,                           /* leangths of arrays*/
    pos,                           /* positions of each array */
    MPI_FLOAT,      /* data type*/
    &upper );
MPI_Type_commit( &upper );

MPI_Recv( a, 1, upper, .... );
```

MPI Derived Data Types

- ***Struct*** constructor is the most general type constructor. The new data type is formed according to completely defined map of the component data types.

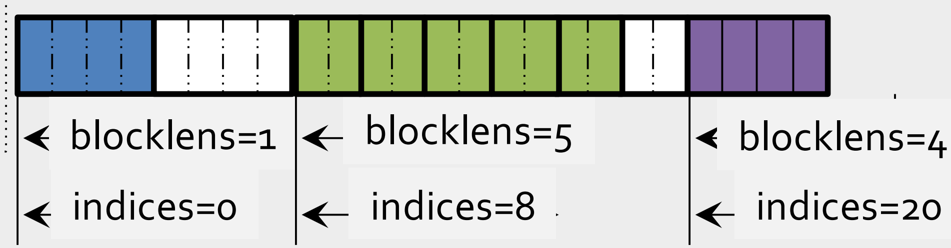
MPI_Type_struct(count, blocklens[], indices[], oldtypes[], newtype)

count: number of blocks,
blocklens: number of elements in each block,
indices: byte displacement of each block,
oldtypes: type of elements in each block,
newtype: new datatype.

```
oldtypes = (
MPI_INT,
MPI_SHORT,
MPI_CHAR
)
```

```
count = 3,
blocklens=[1,5,4]
indices=[0,8,20]
```

newtype



MPI Derived Data Types

```
#include <stddef.h>
struct {   int    i;
             double d[3];
              long   l[8];
              char   c;
} MyStruct;

MyStruct st;
MPI_Datatype myStructType;

int len[5] = { 1, 3, 8, 1, 1 };
MPI_Aint  pos[5] = {offsetof(MyStruct,i), offsetof(MyStruct,d),
                          offsetof(MyStruct,l), offsetof(MyStruct,c),
                           sizeof(MyStruct};
MPI_Datatype typ[5] = {MPI_INT, MPI_DOUBLE, MPI_LONG, MPI_CHAR, MPI_UB};

MPI_Type_struct( 5,  len,  pos,  typ,  &myStructType );
MPI_Type_commit( &myStructType );

MPI_Send( st, 1, myStructType, ... );
```