CPA – Parallel Computing

Degree in Computer Science

S3. Programming with MPI

J. M. Alonso, P. Alonso, F. Alvarruiz, I. Blanquer,
J. Ibáñez, E. Ramos, J. E. Román

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Year 2024/25

Contents

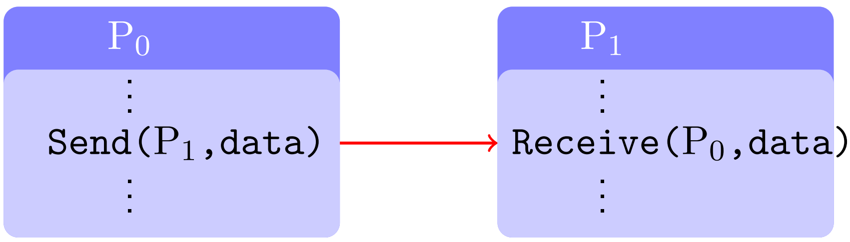
Section 1

Basic Concepts

- Message-Passing Model
- The MPI Standard
- MPI Programming Model

Message-Passing Model

Exchange of information by explicitly sending and receiving messages



Most widely used model in large-scale computing – Software libraries (lower learning curve with respect to a new language)

Advantages:

- Universality
- Easy understanding
- High expressivity
- Higher efficiency

Drawbacks:

- Complex programming
- Total control of communications

The MPI Standard

MPI is a specification proposed by a committee of researchers, users and companies

`https://www.mpi-forum.org`

Specifications:

- MPI-1.0 (1994), last update MPI-1.3 (2008)
- MPI-2.0 (1997), last update MPI-2.2 (2009)
- MPI-3.0 (2012), last update MPI-3.1 (2015)
- MPI-4.0 (2020), last update MPI-4.1 (2021)

Previous approaches:

- Each manufacturer provided its own environment (costly migration)
- PVM (*Parallel Virtual Machine*) constituted a first attempt for standardization

Features of MPI

Main features:

- It is portable to any parallel platform
- It is simple (with only 6 functions any program can be implemented)
- It is powerful (more than 300 API functions)

The standard specifies an interface for C and Fortran

There are many implementations available:

- Proprietary: IBM, Cray, SGI, ...
- MPICH (`www.mpich.org`)
- Open MPI (`www.open-mpi.org`)
- MVAPICH (`mvapich.cse.ohio-state.edu`)

Programming Model

MPI programming is based on library functions
To use them, an initialization is required

Example

```
#include <mpi.h>
int main(int argc, char* argv[]) {
  int k;        /* process rank  */
  int p;        /* number of processes */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &k);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  printf("I am process %d of %d\n", k, p);
  MPI_Finalize();
  return 0;
}
```

- `MPI_Init` and `MPI_Finalize` are compulsory
- Once initialized, different operations can be performed

Programming Model – Operations

Operations can be classified in:

- Point-to-point communication
  Exchange of information between two processes
- Collective communications
  Exchange of information among groups of processes
- Data management
  Derived datatypes (e.g. data stored non-contiguously in memory)
- High-level operations
  Groups, communicators, attributes, topologies
- Advanced operations (MPI-2, MPI-3)
  Input-output, process creation, one-sided communication
- Utilities
  Interaction with the environment

Most communication operations work on communicators

Programming Model – Communicators

A communicator is an abstraction that comprises the following concepts:

- *Group*: group of processes
- *Context*: to avoid interferences among different messages

A communicator groups together p processes

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

Each process has an identifier (rank), a number between 0 and $p-1$

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


Execution Model

The MPI execution model follows a scheme of simultaneous process creation when launching the application

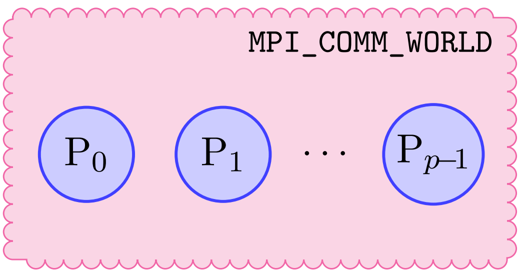
Applications are normally executed by a launcher command

mpiexec -n p program [arguments]

When an application is executed:

- p copies of the same executable are spawn (e.g. with `ssh`)
- A communicator is created (`MPI_COMM_WORLD`) that groups all the processes

MPI-2 provides a mechanism to create new processes



Section 2

Point-to-Point Communication

- Semantics
- Blocking Primitives
- Other Primitives
- Examples

Point-to-Point communication – the Message

Messages must be explicitly issued by the sender and explicitly received by the receiver

Standard send:

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

Standard receive:

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

The content of the message is defined by the first three arguments:

- A memory *buffer* where data is stored
- The message length (number of elements from the buffer)
- Datatype of the elements (e.g. `MPI_INT`)

Point-to-Point Communication – the Envelope

To perform the communication, it is necessary to indicate the destination (`dest`) and the source (`src`)

- The communication is allowed only within the same communicator, `comm`
- The source and destination are specified via process identifiers
- In the reception it is allowed to use `src=MPI_ANY_SOURCE`

An integer number can be used (`tag`) to distinguish among messages of different type

- In the reception it is allowed to use `tag=MPI_ANY_TAG`

In the reception, the status (`stat`) contains information:

- Source process (`stat.MPI_SOURCE`), tag (`stat.MPI_TAG`)
- Message length (explained in p. 43)

Note: pass `MPI_STATUS_IGNORE` if not required

Point-to-Point Send Modes

There are several send modes:

- Synchronous send mode
- Buffered send mode
- Standard send mode

The most commonly used one is the standard mode

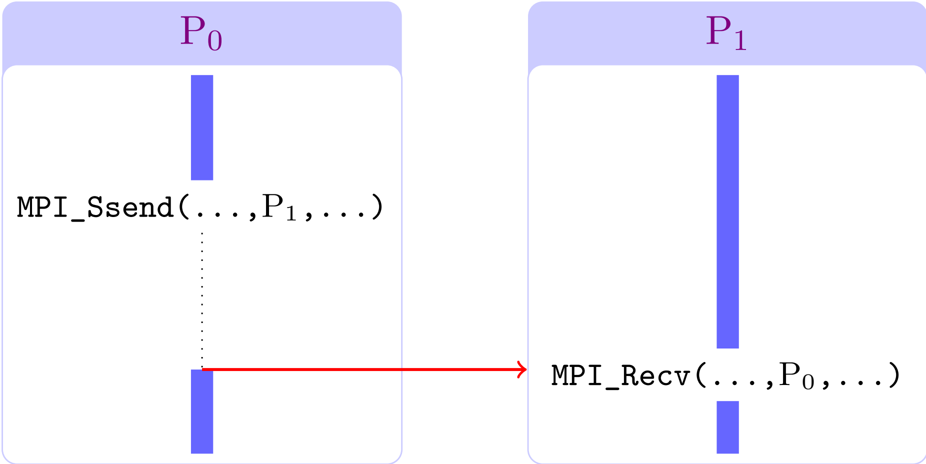
The rest of the modes could be useful to obtain better performance or increased robustness

For each mode, there are blocking and nonblocking primitives

Synchronous Send Mode

```
MPI_Ssend(buf, count, datatype, dest, tag, comm)
```

It implements the send model with "rendezvous": the sender gets blocked until the receiver posts the receive operation

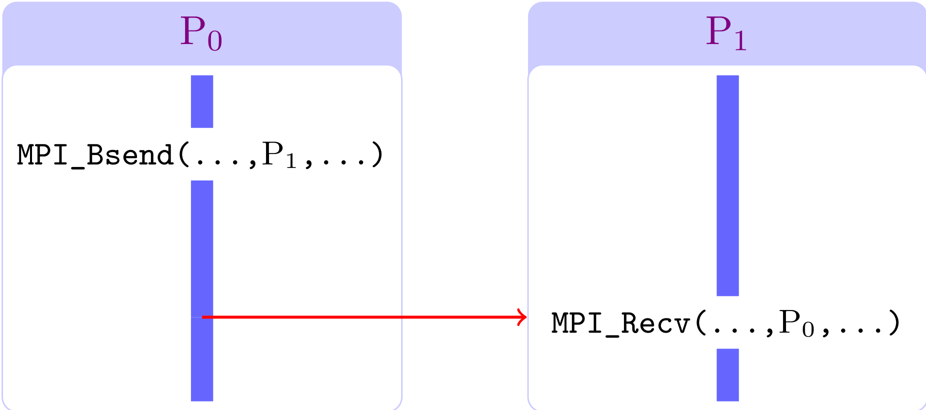


- Inefficient: the sender remains blocked doing no useful work

Buffer-based send Mode

```
MPI_Bsend(buf, count, datatype, dest, tag, comm)
```

The message is copied to an intermediate memory and the sending process continues its execution



- Drawbacks: additional copy and risk of failure
- A buffer may be provided (`MPI_Buffer_attach`)

Standard Send Mode

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

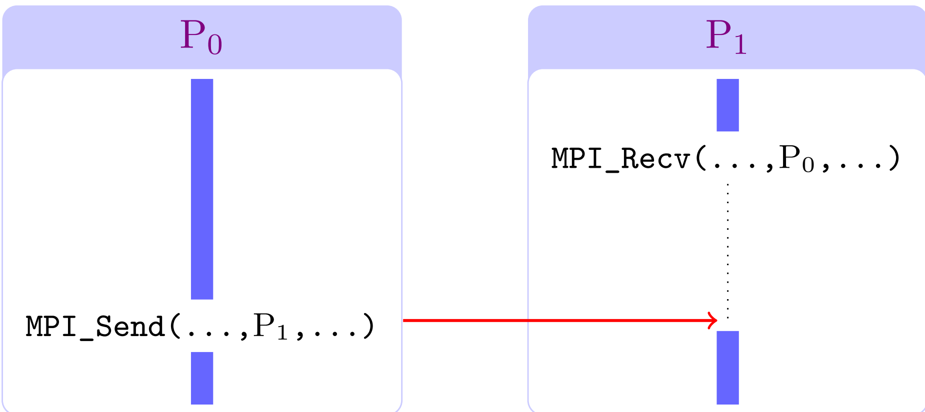
Completion is guaranteed in any kind of systems, since it avoids storage problems

- Short messages are usually sent using `MPI_Bsend`
- Long messages are usually sent with `MPI_Ssend`

Standard Reception

```
MPI_Recv(buf, count, datatype, src, tag, comm, stat)
```

It implements the reception model with "rendezvous": the receiver gets blocked until the message arrives



- Inefficient: the receiver process gets blocked and idle

Nonblocking Send Primitives

```
MPI_Isend(buf, count, datatype, dest, tag, comm, req)
```

The send operation is started, but the sender is not blocked

- It has an additional argument (`req`)
- Before reusing the buffer, one must make sure that the send operation has been completed

Example

```
MPI_Isend(A, n, MPI_DOUBLE, dest, tag, comm, &req);
...
/* Check whether the send operation has finished, with
   MPI_Test or MPI_Wait */
A[10] = 2.6;
```

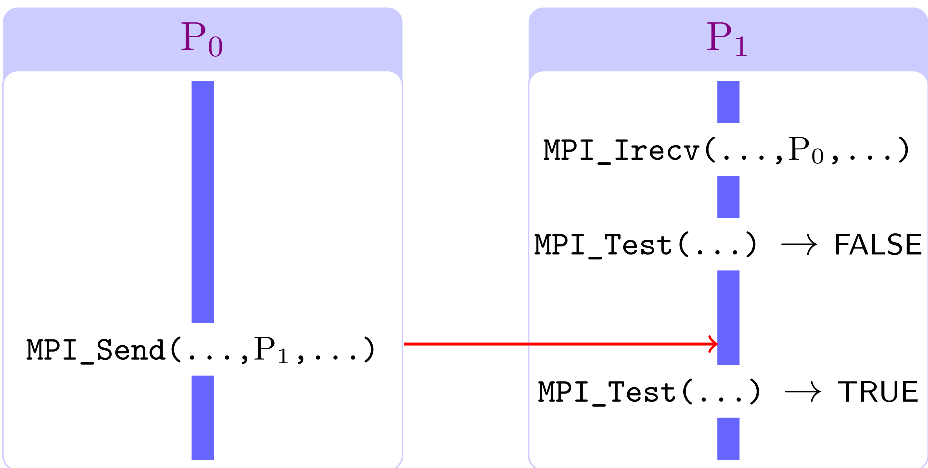
- Overlap communication/computation with no extra copy
- Drawbacks: more complex programming

Nonblocking Reception

```
MPI_Irecv(buf, count, type, src, tag, comm, req)
```

Reception is initiated, but the receiver does not get blocked

- The `stat` argument is replaced by `req`
- It is necessary to check the effective arrival of the message



- Advantage: overlap communication and computation
- Drawback: more complex programming

Combined Operations

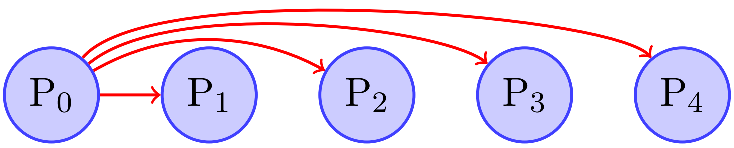
```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
    recvbuf, recvcount, recvtype, source, recvtag, comm,
    status)
```

It performs a send and receive operation in the same call (not necessarily involving the same process)

```
MPI_Sendrecv_replace(buf, count, datatype, dest, sendtag,
    source, recvtag, comm, status)
```

It performs a send and receive operation at the same time on the same variable

Examples – Broadcast



Broadcast of a numeric value from P_0

```
double val;
MPI_Status status;
int p, rank, i;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  read_value(&val);      /* value to be broadcast */
  for (i=1; i<p; i++)
    MPI_Send(&val,1,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
} else {
  MPI_Recv(&val,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
}
```

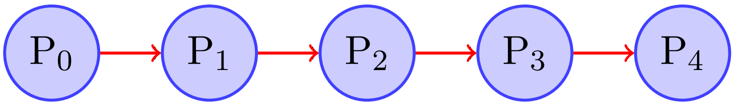
Example – Shift in a 1-D Grid (1)

Each process sends its data to its right neighbor and substitutes it by the data received from its left neighbor

Shift in a 1-D Grid - trivial version

```
if (rank == 0) {
  MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
} else if (rank == p-1) {
  MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
} else {
  MPI_Send(&val, 1, MPI_DOUBLE, rank+1, 0, comm);
  MPI_Recv(&val, 1, MPI_DOUBLE, rank-1, 0, comm, &status);
}
```

Drawback: Sequentialization - communications are (probably) performed sequentially, without concurrency



Example – Shift in a 1-D Grid (2)

In some cases, programming can be simplified using null processes

Shift in a 1-D Grid – null processes

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

Sending to process `MPI_PROC_NULL` returns immediately; reception of a message from process `MPI_PROC_NULL` returns immediately without receiving anything

This version does not solve the problem of sequentialization

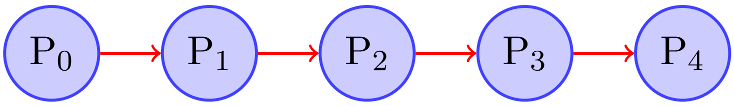
Example – Shift in a 1-D Grid (3)

Solution to sequentialization: Odd-Even Protocol

Shift in a 1-D Grid – odd-even protocol

```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

if (rank%2 == 0) {
  MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
  MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
} else {
  MPI_Recv(&tmp, 1, MPI_DOUBLE, prev, 0, comm, &status);
  MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
  val = tmp;
}
```



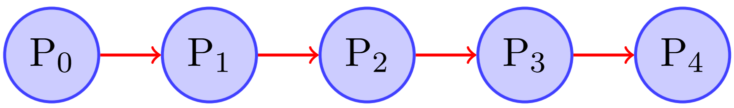
Example – Shift in a 1-D Grid (4)

Solution to sequentialization: Combined Operations

Shift in a 1-D Grid – sendrecv

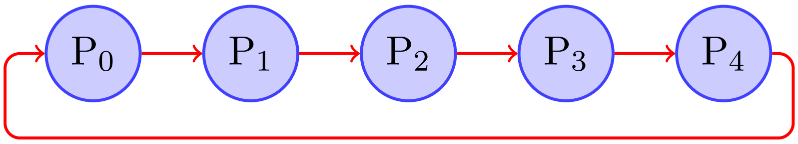
```
if (rank == 0) prev = MPI_PROC_NULL;
else prev = rank-1;
if (rank == p-1) next = MPI_PROC_NULL;
else next = rank+1;

MPI_Sendrecv_replace(&val, 1, MPI_DOUBLE, next, 0, prev, 0,
                     comm, &status);
```



Example – Shift around a Ring

In the case of a ring, all processes must send and receive



Shift around a ring – trivial version

```
if (rank == 0) prev = p-1;
else prev = rank-1;
if (rank == p-1) next = 0;
else next = rank+1;

MPI_Send(&val, 1, MPI_DOUBLE, next, 0, comm);
MPI_Recv(&val, 1, MPI_DOUBLE, prev, 0, comm, &status);
```

A deadlock will happen in the case of synchronous send
Solutions: odd-even protocol or combined operations

Section 3

Collective Communication

- Synchronization
- Broadcast
- Scatter
- Reduction

Collective Communication Operations

They involve all processes in a group (communicator) – all of them must execute the operation

Available operations:

- Synchronization (*Barrier*)
- Broadcast (*Bcast*)
- Distribution (*Scatter*)
- Concatenation (*Gather*)
- Conc.-broadcast (*Allgather*)
- All-to-all comm. (*Alltoall*)
- Reduction (*Reduce*)
- Prefix reduction (*Scan*)

These operations usually take as an argument a process (`root`) who plays a special role

"`All`" prefix: Every process receives the result

"`v`" suffix: The size of data received or sent by each process may be different

Synchronization

`MPI_Barrier(comm)`

Pure synchronization operation

- All the processes of the communicator `comm` wait until all of them have reached this call

Example – time measurement

```
MPI_Barrier(comm);
t1 = MPI_Wtime();
/*
    ...
*/
MPI_Barrier(comm);
t2 = MPI_Wtime();

if (!rank) printf("Elapsed time: %f s.\n", t2-t1);
```

Broadcast

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

The `root` process broadcasts the message indicated by the first three arguments to the rest of processes

	Initial State		*Final State*
P_0			A
P_1			A
P_2	A	$\longrightarrow$	A
P_3			A
P_4			A
P_5			A

Distribution (Scatter)

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm)
```

The `root` process distributes a series of consecutive fragments of the buffer to the rest of processes (including itself)

			Initial State					*Final State*
P_0								A
P_1								B
P_2	A	B	C	D	E	F	$\longrightarrow$	C
P_3								D
P_4								E
P_5								F

Asymmetric version: `MPI_Scatterv`

Distribution: Example

Process P_0 scatters a vector of 15 elements (a) to 3 processes that receive data in vector b

Scatter example

```
int main(int argc, char *argv[])
{
  int i, myproc;
  int a[15], b[5];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myproc);
  if (myproc==0) for (i=0;i<15;i++) a[i] = i+1;

  MPI_Scatter(a, 5, MPI_INT, b, 5, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```


Concatenation (Gather)

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm)
```

It is the reverse operation of `MPI_Scatter`: Each process sends a message to `root`, who stores them in an ordered way according to the index of the process owning each fragment

	Initial State		*Final State*					
P_0	A							
P_1	B							
P_2	C	$\longrightarrow$	A	B	C	D	E	F
P_3	D							
P_4	E							
P_5	F							

Asymmetric version: `MPI_Gatherv`

Concatenation-Broadcast (Allgather)

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf,
              recvcount, recvtype, comm)
```

Similar to the `MPI_Gather` operation, but in this case all processes get the result

Initial State

P_0	A
P_1	B
P_2	C
P_3	D
P_4	E
P_5	F

$\longrightarrow$

Final State

A	B	C	D	E	F
A	B	C	D	E	F
A	B	C	D	E	F
A	B	C	D	E	F
A	B	C	D	E	F
A	B	C	D	E	F

Asymmetric version: `MPI_Allgatherv`

All-to-all Communication

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm)
```

An extension of `MPI_Allgather`, where each process sends different data and receives (ordered) data from the rest

Initial State

P_0	A_0	A_1	A_2	A_3	A_4	A_5
P_1	B_0	B_1	B_2	B_3	B_4	B_5
P_2	C_0	C_1	C_2	C_3	C_4	C_5
P_3	D_0	D_1	D_2	D_3	D_4	D_5
P_4	E_0	E_1	E_2	E_3	E_4	E_5
P_5	F_0	F_1	F_2	F_3	F_4	F_5

$\longrightarrow$

Final State

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

Asymmetric version: `MPI_Alltoallv`

Reduction

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

Similar to `MPI_Gather`, but instead of concatenation, an arithmetic or logic operation is applied to the data (sum, max, and, ..., or user-defined)

The final result is stored in the `root` process

	Initial State		*Final State*
P_0	A		
P_1	B		
P_2	C	$\longrightarrow$	A+B+C+D+E+F
P_3	D		
P_4	E		
P_5	F		

Multi-Reduction

```
MPI_Allreduce(sendbuf, recvbuf, count, type, op, comm)
```

Extension of `MPI_Reduce` in which all processes get the result

Scalar product of vectors

```
double par_dot(double local_x[],double local_y[],int local_n)
{
  double local_dot;
  double dot;

  local_dot = seq_dot(local_x, local_y, local_n);
  MPI_Allreduce(&local_dot, &dot, 1, MPI_DOUBLE,
                MPI_SUM, MPI_COMM_WORLD);
  return dot;
}
```

Prefix Reduction (Scan)

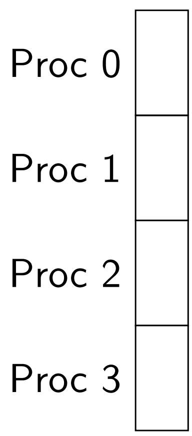
```
MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)
```

Extension of the reduction operations in which each process receives the result of processing all the elements from process 0 to itself

	Initial State		*Final State*
P_0	A		A
P_1	B		A+B
P_2	C	$\longrightarrow$	A+B+C
P_3	D		A+B+C+D
P_4	E		A+B+C+D+E
P_5	F		A+B+C+D+E+F

Example of Prefix Reduction

Given a vector of length N and distributed among the processes, where each process has n_{local} consecutive elements of the vector, we want to compute the initial position of the local subvector



Computation of the initial index of a parallel vector

```
int rstart, nlocal, N;

compute_nlocal(N,&nlocal);    /* e.g. nlocal=N/p */
MPI_Scan(&nlocal,&rstart,1,MPI_INT,MPI_SUM,comm);
rstart -= nlocal;
```

Section 4

Other Functionalities

- Derived Datatypes

Basic Datatypes

The basic datatypes in the C language are:

`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed 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`

- In Fortran, there are similar definitions
- Along with the previous ones, there are the special types `MPI_BYTE` and `MPI_PACKED`

Multiple Data

It is possible to send/receive multiple data in one message:

- The sender indicates the number of data to be sent using argument `count`
- The message is formed by the first `count` elements contiguous in memory
- In the receiver, argument `count` indicates the buffer size – the actual size of the received message can be obtained with:

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

This approach is not intended for:

- Composing a message with data of different types
- Sending non-contiguous data, even of the same type

Derived Datatypes

MPI allows defining new types from other types

The procedure goes through the following steps:

1. The programmer defines the new type, indicating:
   - The datatypes of its different constituents
   - The number for elements of each type
   - The relative displacements for each element
2. It is registered as a new MPI datatype (`commit`)
3. From then on, it can be used in any communication operation as it was a basic datatype
4. If no longer needed, the type must be destroyed (`free`)

Advantages:

- It simplifies programming when it is used several times
- There is no intermediate copy, since data is compressed only at the time of sending

Regular Derived Datatypes

```
MPI_Type_vector(count, length, stride, type, newtype)
```

It creates an homogeneous datatype from the evenly distributed elements of an array

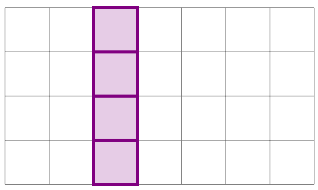
1. How many blocks are included (`count`)
2. Which is the length of the blocks (`length`)
3. Which is the separation from an element of a block and the same element in the next block (`stride`)
4. Of which type are the individual blocks (`type`)

Related constructors:

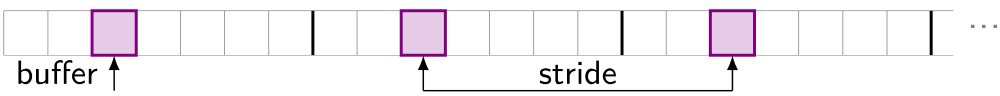
- `MPI_Type_contiguous`: contiguous elements
- `MPI_Type_indexed`: variable length and displacement

Regular Derived Datatypes: Example

We want to send a *column* of a matrix `A[4][7]`



In C, bidimensional arrays are stored by rows




```
double A[4][7];
MPI_Datatype column;
MPI_Type_vector(4, 1, 7, MPI_DOUBLE, &column);
MPI_Type_commit(&column);
if (my_rank == 0) {           /* sends the 3rd column */
  MPI_Send(&A[0][2], 1, column, 1, 0, comm);
} else {
  MPI_Recv(&A[0][2], 1, column, 0, 0, comm, &status);
}
```