Parallel and Distributed Programming

Imad Kissami[1]

[1]Mohammed VI Polytechnic University, Benguerir, Morocco

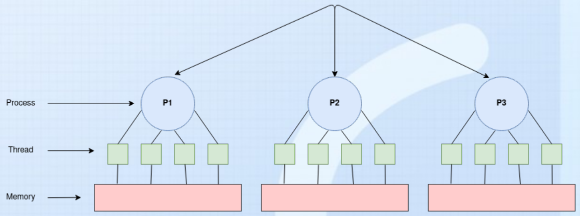
October 30, 2025

Outline of this lecture

- Distributed Memory Architectures & MPI
- Point-to-point communications
- Collective communications
- Communication modes
- Derivatives types
- Communicators

Distributed Memory Architecture

Distributed Memory Multiprocessors



- Each processor has a local memory
- Processors must communicate to access non-local data
- Parallel applications must be partitioned across
  - Processors: execution units
  - Memory: data partitioning

Distributed Memory Architecture

Message passing programming model

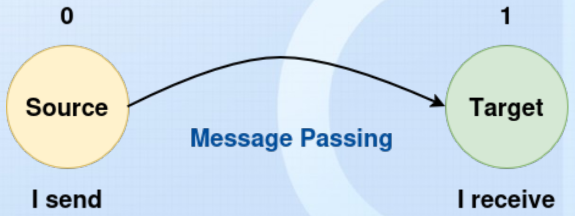
In the message-passing model:

- The program is written in a conventional language (e.g., C, Fortran).
- Variables are private and reside in each process's local memory.
- Communication between processes occurs via explicit message-passing subroutine calls.

Distributed Memory Architecture

Message passing concepts

If a message is sent to a process, the process must receive it.

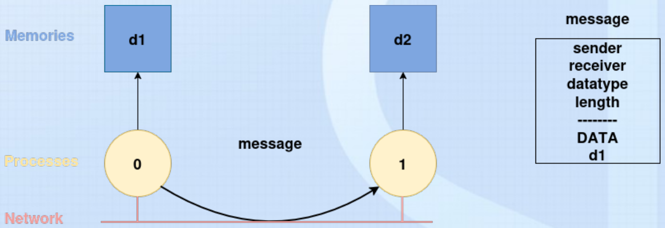


Distributed Memory Architecture

Message content

A message comprises:

- Data chunks passing from the sending process to the receiving process/pocesses (e.g., scalar variables, arrays).
- Metadata, including:
  - ID of the sending process, Data type and length, ID of the receiving process.



Message Construction

Distributed Memory Architecture

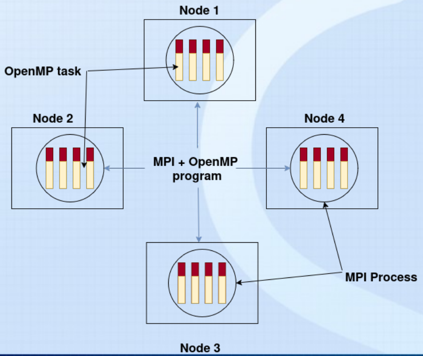
Environment

- The exchanged messages are interpreted and managed by an environment comparable to telephony, e-mail, postal mail, etc.
- The message is sent to a specified address.
- The receiving process must be able to classify and interpret the messages which are sent to it.
- The environment in question is MPI (Message Passing Interface). An MPI application is a group of autonomous processes, each executing its own code and com-communicating via calls to MPI library subroutines.

Distributed Memory Architecture

MPI vs. OpenMP

- MPI uses a distributed memory model.
- OpenMP uses a shared memory model.



Distributed Memory Architecture

History

- `MPI 1.0 (June 1994):` Initial definition.
- `MPI 2.0 (1997):` Added features like parallel I/O.
- `MPI 3.0 (2012):` Introduced nonblocking collective communication.
- `MPI 4.0 (2021):` Added large count, partitioned communication.
- `MPI 4.1 (2023):` Clarifications and small extensions to MPI 4.0, including improved partitioned communication semantics, new predefined reduction operations, and clarifications for persistent collective operations.
- `MPI 5.0 (June 2025):` Major update introducing a standard Application Binary Interface (ABI) for interoperability among MPI implementations, expanded language bindings (including modern C++ support), improved asynchronous and persistent communication models, new tools interfaces, and further refinements to partitioned communication and large-count operations.

Distributed Memory Architecture

`MPI_Init` & `MPI_Finalize`

- Every program unit calling MPI routines has to include the header file mpi.h.
- Initialize MPI environment: `MPI_Init()`

```
int MPI_Init(int *argc, char ***argv);
```

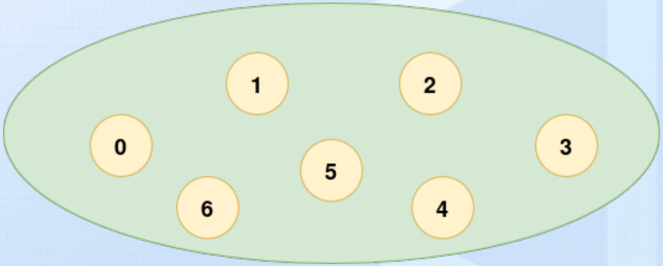
- Finalize MPI environment: `MPI_Finalize()`

```
int MPI_Finalize(void)
```

Distributed Memory Architecture

Communicators

- All the MPI operations occur in a defined set of processes, called `communicator`.
- The default `communicator` is `MPI_COMM_WORLD`, which includes all the active processes.



MPI_COMM_WORLD Communicator

Distributed Memory Architecture

Termination of a program

Sometimes, a program encounters some issue during its execution and has to stop prematurely. For example, we want the execution to stop if one of the processes cannot allocate the memory needed for its calculation. In this case, we call the `MPI_Abort()` subroutine instead of the Fortran instruction stop (Or exit in C).

```
int MPI_Abort(MPI_Comm comm, int error)
```

- `comm`: the communicator of which all the processes will be stopped ; it is advised to use `MPI_COMM_WORLD` in general.
- `error` : the error number returned to the UNIX environment

The MPI library

Rank and size

- One can access to the number of processes managed by a given communicator using the `MPI_Comm_size()` function

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

- You can get the rank of a process, within a communicator, by calling `MPI_Comm_rank()` function

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

The MPI library

Rank and size Example

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

int main(int argc, char *argv[]) {
   int rank, size;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("I am process %d among %d\n", rank, size);

   MPI_Finalize();
   return 0;
}
```

```
$ mpicc -o who_am_I who_am_I.c
```

```
$ mpirun -n 2 who_am_I
I am the proccess 0 among 2
I am the proccess 1 among 2
```

Point-to-point Communications

Blocking Send `MPI_SEND`

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

- Sending, from the address **`buf`**, a message of **`count`** elements of type **`datatype`**, tagged **`tag`**, to the process of rank **`dest`** in the communicator **`comm`**.
- the execution remains blocked until the message can be re-written without risk of overwriting the value to be sent. In other words, the execution is blocked as long as the message has not been received.

Point-to-point Communications

Blocking Receive MPI_RECV

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

- Receiving, at the address buf, a message of count elements of type datatype, tagged tag, from the process of rank source in the communicator comm.
- status_msg stores the state of a receive operation : source, tag, code, ...
- An MPI_RECV can only be associated to an MPI_SEND if these two calls have the same envelope (source, dest, tag, comm).
- the execution remains blocked until the message content corresponds to the received message.

Point-to-point Communications

Blocking Send / Receive Full example

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

int main(int argc, char *argv[]) {
    int rank, value, tag = 100;
    MPI_Status status;

    MPI_Init(&argc, &argv); // Initialize MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get process rank

    if (rank == 2) {
        value = 1000;
        MPI_Send(&value, 1, MPI_INT, 5, tag, MPI_COMM_WORLD);
    } else if (rank == 5) {
        MPI_Recv(&value, 1, MPI_INT, 2, tag, MPI_COMM_WORLD, &status);
        printf("I, process 5, received %d from process 2.\n", value);
    }
    MPI_Finalize(); // Finalize MPI environment
    return 0;
}
```

```
mpirun -n 6 point_to_point
I, process 5, I received  1000  from process 2.
```

Point-to-point Communications

C MPI Datatypes

MPI Datatype	C Datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	Raw byte data

Table: Mapping of MPI datatypes to C datatypes

Point-to-point Communications

Other possibilities

- Wildcard Parameters
  - The sender process rank can be replaced with `MPI_ANY_SOURCE`.
  - The message tag can be replaced with `MPI_ANY_TAG`.
- Dummy Processes
  - Communications with the process rank `MPI_PROC_NULL` have no effect.
- Ignoring Status
  - Use the predefined constant `MPI_STATUS_IGNORE` instead of the status variable if the status information is not needed.
- Derived Datatypes
  - You can send more complex data structures by creating derived datatypes.
- Simultaneous Send and Receive
  - Use operations like `MPI_Sendrecv()` or `MPI_Sendrecv_replace()` for simultaneous send and receive actions.

Point-to-point Communications

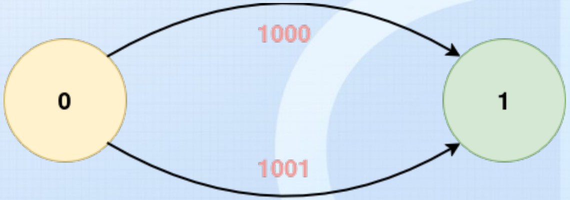
Simultaneous send and receive MPI_SENDRECV

```
int MPI_Sendrecv(const 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_msg)
```

- Sending, from the address sendbuf, a message of sendcount elements of type sendtype, tagged sendtag, to the process dest in the communicator comm ;
- Receiving, at the address recvbuf, a message of recvcount elements of type recvtype, tagged recvtag, from the process source in the communicator comm.
- Here, the receiving zone recvbuf must be different from the sending zone sendbuf.

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV



sendrecv Communication between the Processes 0 and 1

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV: Full example

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

int main(int argc,char *argv[]) {
    int rank,value, num_proc, message, tag=110;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    num_proc=(rank+1)%2;
    message = rank+1000;
    MPI_Sendrecv(&message,1,MPI_INT,num_proc,tag,&value,1,MPI_INT,
    num_proc,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    printf("I, process %d, I received %d from process %d.\n", rank,value,num_proc);

    MPI_Finalize();
}
```

```
mpirun -n 2 simultaneoussendrecv
I, process 0, I received 1001 from the process 1.
I, process 1, I received 1000 from the process 0.
```

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV: Remarks

In the case of a synchronous implementation of the `MPI_SEND()` subroutine, if we replace the `MPI_SENDRECV()` subroutine in the example above by `MPI_SEND()` followed by `MPI_RECV()` , the code will deadlock. Indeed, each of the two processes will wait for a receipt confirmation, which will never come because the two sending operations would stay suspended.

```
val = rank+1000;
MPI_Send(&val,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD);
MPI_Recv(value,1,MPI_INT,num_proc,tag,MPI_COMM_WORLD,&statut);
```

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV_REPLACE

```
int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int ↩
        source, int recvtag, MPI_Comm comm, MPI_Status *status)
```

- Sending, from the address `buf`, a message of count elements of type `datatype`, tagged `sendtag`, to the process `dest` in the communicator comm ;
- Receiving a message at the same address, with same count elements and same datatype, tagged `recvtag`, from the process `source` in the communicator `comm`.
- Contrary to the usage of MPI_SENDRECV , the receiving zone is the same here as the sending zone `buf`.

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV_REPLACE : full example

```
int main(int argc, char *argv[]) {
    int rank, tag = 11, m = 2;
    int A[m][m];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        A[0][0] = 1; A[0][1] = 2; A[1][0] = 3; A[1][1] = 4;
        MPI_Send(A, 2, MPI_INT, 1, tag, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&(A[0][1]), 2, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);

        printf("I process %d, received 2 elements from process %d with tag %d. "
        "The elements are %d %d.\n", rank, status.MPI_SOURCE, status.MPI_TAG, A[0][1], A[1][0]);
    }
    MPI_Finalize();
}
```

Point-to-point Communications

Simultaneous send and receive MPI_SENDRECV_REPLACE : full example

```
mpirun -n 2 ./simultaneoussendrecv_replace

I process 1, received 2 elements from process 0 with tag 11. The elements are 1 2.
```

Collective Communications

General concepts

- Collective communications allow making a series of point-to-point communications in one single call.
- A collective communication always concerns all the processes of the indicated communicator.
- For each process, the call ends when its participation in the collective call is completed, in the sense of point-to-point communications (therefore, when the concerned memory area can be changed).
- The management of tags in these communications is transparent and system-dependent. Therefore, they are never explicitly defined during calls to subroutines. An advantage of this is that collective communications never interfere with point-to-point communications.

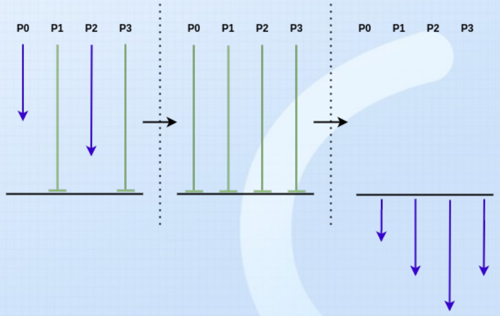
Collective Communications

Types of collective communications

- One which ensures global synchronizations : MPI_Barrier()
- Ones which only transfer data :
  - Global distribution of data: `MPI_Bcast()`
  - Selective distribution of data: `MPI_Scatter()`
  - Collection of distributed data: `MPI_Gather()`
  - Collection of distributed data by all the processes: `MPI_Allgather()`
  - Collection and selective distribution by all the processes of distributed data: `MPI_Alltoall()`
- Ones which, in addition to the communications management, carry out operations on the transferred data :
  - Reduction operations (sum, product, maximum, minimum, etc.), whether of a predefined or personal type: `MPI_Reduce()`
  - Reduction operations with distributing of the result (this is in fact equivalent to an `MPI_Reduce()` followed by an `MPI_Bcast()`): `MPI_Allreduce()`

Collective Communications

Global synchronization: MPI_Barrier()



Global Synchronization : MPI_BARRIER()

```
int MPI_Barrier(MPI_Comm comm)
```

Collective Communications

Global distribution: MPI_BCAST()

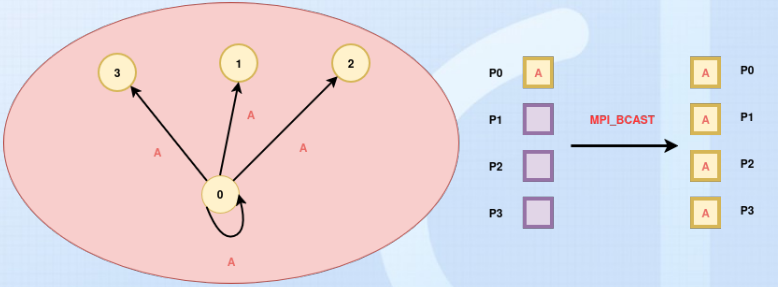


Figure: Global distribution : MPI_Bcast()

Collective Communications

Global distribution: MPI_Bcast()

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

- Send, starting at position `buffer`, a message of `count` element of type `datatype`, by the `root` process, to all the members of communicator `comm`.
- Receive this message at position `buffer` for all the processes other than the `root`.

Collective Communications

Global distribution: MPI_Bcast() Full example 1

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

int main(int argc, char *argv[]) {
   int rank, value;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 2) value = rank+1000

   MPI_Bcast(&value,1,MPI_INT,2,MPI_COMM_WORLD);

   printf("I, process %d, received %d of process 2\n", rank,value);

   MPI_Finalize();
}
```

```
mpirun -n 3 bcast

I, process 2, received 1002 of process 2
I, process 1, received 1002 of process 2
I, process 0, received 1002 of process 2
```

Collective Communications

Selective distribution: MPI_Scatter()

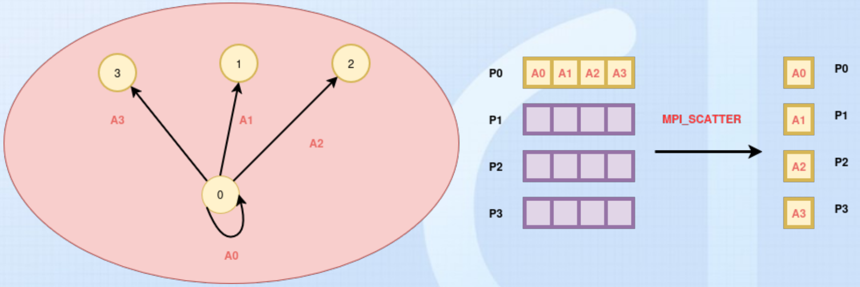


Figure: Selected distribution : MPI_Scatter()

Collective Communications

Selective distribution: MPI_Scatter()

```
int MPI_Scatter(const void *sendbuf,int sendcount, MPI_Datatype sendtype,void *recvbuf, int ↩
     recvcount,MPI_Datatype recvtype, int root,MPI_Comm comm)
```

- Scatter by process `root`, starting at position `sendbuf`, message `sendcount` element of type `sendtype`, to all the processes of communicator `comm`.
- Receive this message at position `recvbuf`, of `recvcount` element of type `recvtype` for all processes of communicator `comm`.
- ➠ The couples `(sendcount, sendtype)` and `(recvcount, recvtype)` must represent the same quantity of data.
- ➠ Data are scattered in chunks of same size ; a chunk consists of `sendcount` elements of type `sendtype`.
- ➠ The i-th chunk is sent to the i-th process.

Collective Communications

Selective distribution: MPI_SCATTER() Full example

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

int main(int argc, char *argv[]) {
   int nb_values = 8, rank, nb_procs, block_length, i;
   float *values, *recvdata;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   block_length = nb_values / nb_procs;
   recvdata = (float *)malloc(block_length * sizeof(float));

   if (rank == 2) {
      values = (float *)malloc(nb_values * sizeof(float));
      for (i = 0; i < nb_values; i++) values[i] = 1001.0 + i;

      printf("Process %d sends the values array: ", rank);
      for (i = 0; i < nb_values; i++) printf("%f ", values[i]);

      printf("\n");
   }
```

Collective Communications

Selective distribution: MPI_SCATTER() Full example

```
MPI_Scatter(values, block_length, MPI_FLOAT, recvdata, block_length, MPI_FLOAT, 2, ↩
       MPI_COMM_WORLD);

printf("Process %d received: ", rank);
for (i = 0; i < block_length; i++) {
    printf("%f ", recvdata[i]);
}
printf("from process 2\n");

MPI_Finalize();
return 0;
}
```

```
mpirun -n 4 scatter
I, process 2 send the values array : 1001. 1002. 1003. 1004. 1005. 1006. 1007. 1008.
I, process 0, received 1001. 1002. from processus 2
I, process 1, received 1003. 1004. from processus 2
I, process 3, received 1007. 1008. from processus 2
I, process 2, received 1005. 1006. from processus 2
```

Collective Communications

Selective collection: MPI_Gather()

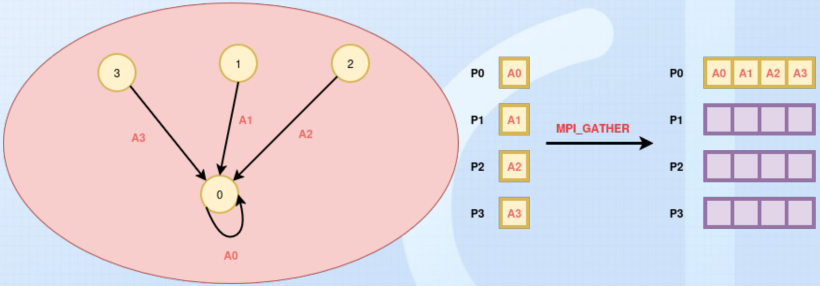


Figure: Collection : MPI_Gather()

Collective Communications

Selective collection: MPI_Gather()

```
int MPI_Gather(const void*sendbuf,int sendcount, MPI_Datatype sendtype,void *recvbuf, int ↩
     recvcount,MPI_Datatype recvtype, int root,MPI_Comm comm)
```

- Send for each process of communicator `comm`, a message starting at position `sendbuf`, of `sendcount` element type `sendtype`
- Collect all these messages by the `root` process at position `recvbuf`, `recvcount` element of type `recvtype`.

➠ The couples `(sendcount, sendtype)` and `(recvcount, recvtype)` must represent the same size of data.

➠ The data are collected in the order of the process ranks.

Collective Communications

Selective collection: MPI_GATHER() Full example

```c
//gather.c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int nb_values=6,rank,nb_procs,block_length,i;
    float recvdata[nb_values],*values;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    block_length = nb_values/nb_procs;

    values = (float *)malloc(block_length * sizeof(float));
    for (i = 0; i < block_length; i++) values[i]=1001.+rank*block_length+i;

    printf("I, process %d sent my values array : ",rank);
```

Collective Communications

Selective collection: MPI_GATHER() Full example

```c
    for (i=0; i<block_length;i++) {printf("%f ",values[i]);} printf("\n");

    MPI_Gather(values, block_length, MPI_FLOAT, recvdata, block_length, MPI_FLOAT, 0, ↩
          MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Root process gathered data: ");
        for (i = 0; i < nb_procs * block_length; i++) {
            printf("%f ", recvdata[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

```
mpirun -n 3 gather
I, process 0 sent my values array : 1001.000000 1002.000000
I, process 1 sent my values array : 1003.000000 1004.000000
I, process 2 sent my values array : 1005.000000 1006.000000
Root process gathered data: 1001.000000 1002.000000 1003.000000 1004.000000 1005.000000 \
      1006.000000
```

Collective Communications

Global collection: MPI_Allgather()

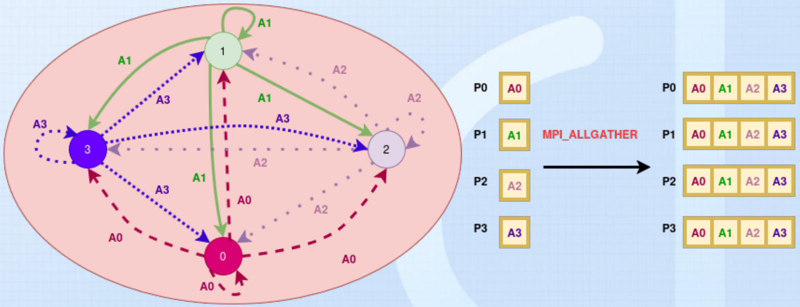


Figure: Gather-to-all : MPI_Allgather()

Collective Communications

Global collection: MPI_Allgather()

```
int MPI_Allgather(const void *sendbuf,int sendcount, MPI_Datatype sendtype,void *recvbuf, int ↩
     recvcount,MPI_Datatype recvtype, MPI_Comm comm)
```

- Corresponds to an `MPI_Gather()` followed by an `MPI_Bcast()`
- Send by each process of communicator `comm`, a message starting at position `sendbuf`, of `sendcount` element, type `sendtype`.
- Collect all these messages, by all the processes, at position `recvbuf` of `recvcount` element type `recvtype`.

➠ The couples `(sendcount, sendtype)` and `(recvcount, recvtype)` must represent the same data size.

➠ The data are gathered in the order of the process ranks.

Collective Communications

Global collection: MPI_Allgather() Full example

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

int main(int argc, char *argv[]) {
    int nb_values = 6, rank, nb_procs, block_length, i;
    float recvdata[nb_values], *values;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    block_length = nb_values / nb_procs;
    values = (float *)malloc(block_length * sizeof(float));

    for (i = 0; i < block_length; i++) values[i] = 1001. + rank * block_length + i;

    MPI_Allgather(values, block_length, MPI_FLOAT, recvdata, block_length, MPI_FLOAT, ↩
          MPI_COMM_WORLD);

   /* print output */
    MPI_Finalize();
}
```

Collective Communications

Global collection: MPI_Allgather() Full example

```
mpirun -n 3 allgather

I, process 0, received 1001.000000 1002.000000 1003.000000 1004.000000 1005.000000 1006.000000
I, process 1, received 1001.000000 1002.000000 1003.000000 1004.000000 1005.000000 1006.000000
I, process 2, received 1001.000000 1002.000000 1003.000000 1004.000000 1005.000000 1006.000000
```

Collective Communications

Selective collection: MPI_Gatherv()

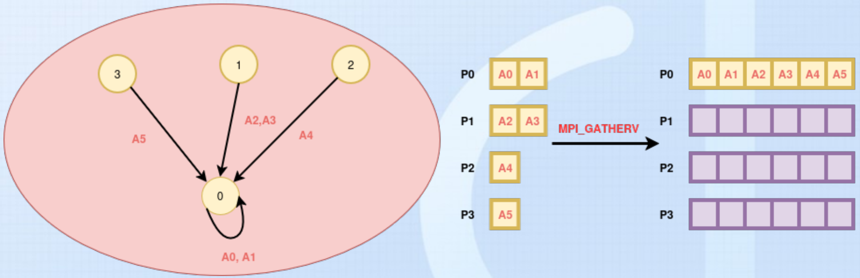


Figure: Extended gather : MPI_Gatherv()

Collective Communications

Selective collection: MPI_Gatherv()

```
int MPI_Gatherv(const void *sendbuf,int sendcount,MPI_Datatype sendtype, void *recvbuf,const int *↩
    recvcounts,const int *displs, MPI_Datatype recvtype,root,MPI_Comm comm)
```

- This is an `MPI_Gather()` where the size of mess. can be different among procs
- The i-th proc of the communicator `comm` sends to `root`, a message starting at position `sendbuf`, of `sendcount` element of type `sendtype`, and receives at position `recvbuf`, of `recvcounts(i)` element of type `recvtype`, with a displacement of `displs(i)`

➠ The couples `(sendcount,sendtype)` of the i-th process and `(recvcounts(i), recvtype)` of process `root` must be such that the data size sent and received is the same.

Collective Communications

Selective collection: MPI_Gatherv() Full example

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

int main(int argc, char *argv[]) {
   int nb_values = 10, rank, nb_procs, block_length, remainder, i;
   float recvdata[nb_values];
   float *values;
   int *nb_elements_received, *displacement;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   block_length = nb_values / nb_procs;
   remainder = nb_values % nb_procs;
   if (rank < remainder) block_length = block_length + 1;

   values = (float *)malloc(block_length * sizeof(float));
   for (i = 0; i < block_length; i++)
   values[i] = 1001. + rank * (nb_values / nb_procs) + (rank < remainder ? rank : remainder) + i;
```

Collective Communications

Selective collection: MPI_Gatherv() Full example

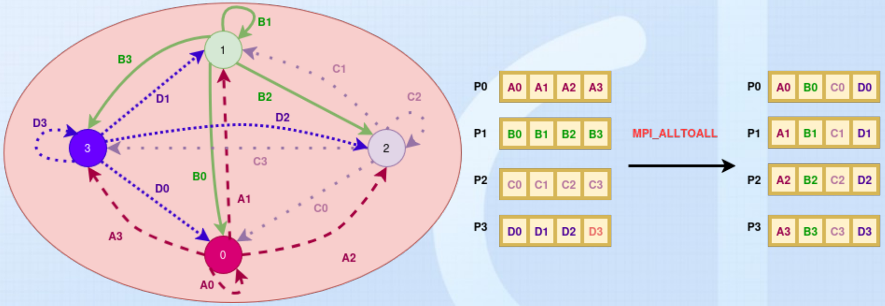
```c
    printf("I, process %d send my values array: ", rank);
    for (i = 0; i < block_length; i++) {
        printf("%f ", values[i]);
    }
    printf("\n");

    if (rank == 2) {
        nb_elements_received = (int *)malloc(nb_procs * sizeof(int));
        displacement = (int *)malloc(nb_procs * sizeof(int));

        nb_elements_received[0] = nb_values / nb_procs;
        if (remainder > 0) nb_elements_received[0] = nb_elements_received[0] + 1;

        displacement[0] = 0;
        for (i = 1; i < nb_procs; i++) {
            displacement[i] = displacement[i - 1] + nb_elements_received[i - 1];
            nb_elements_received[i] = nb_values / nb_procs;
            if (i < remainder) nb_elements_received[i] = nb_elements_received[i] + 1;
        }
    }

    MPI_Finalize();
}
```

Selective collection: MPI_Gatherv() Full example

```
mpirun -n 4 gatherv

I, process 2 send my values array: 1007.000000 1008.000000
I, process 3 send my values array: 1009.000000 1010.000000
I, process 1 send my values array: 1004.000000 1005.000000 1006.000000
I, process 0 send my values array: 1001.000000 1002.000000 1003.000000
```

Collective Communications

Global collection & distribution: MPI_Alltoall()



P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

P0	A0	B0	C0	D0
P1	A1	B1	C1	D1
P2	A2	B2	C2	D2
P3	A3	B3	C3	D3

Figure: Collection and distribution : : MPI_Alltoall()

Collective Communications

Global collection & distribution: MPI_Alltoall()

```
int MPI_Alltoall(const void *sendbuf,int sendcount, MPI_Datatype sendtype,void *recvbuf, int ↩
     recvcount,MPI_Datatype recvtype, MPI_Comm comm)
```

- the i-th process sends its j-th chunk to the j-th process which places it in its i-th chunk.

➠ The couples (`sendcount, sendtype`) and (`recvcount, recvtype`) must be such that they represent equal data sizes.

Collective Communications

Global collection & distribution: MPI_Alltoall() Full example

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

int main(int argc, char *argv[]) {
    int nb_values = 8;
    int rank, nb_procs, block_length, i;
    int recvdata[nb_values], values[nb_values];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i = 0; i < nb_values; i++)
    values[i] = 1001. + rank * nb_values + i;

    block_length = nb_values / nb_procs;

    printf("I, process %d sent my values array: ", rank);

    for (i = 0; i < nb_values; i++) printf("%d ", values[i]);
    printf("\n");
```

Collective Communications

Global collection & distribution: MPI_Alltoall() Full example

```c
    MPI_Alltoall(values, block_length, MPI_INT, recvdata, block_length, MPI_INT, MPI_COMM_WORLD);

    printf("I, process %d, received: ", rank);

    for (i = 0; i < nb_values; i++) printf("%d ", recvdata[i]);
    printf("\n");

    MPI_Finalize();
    return 0;
}
```

```
mpirun -n 4 alltoall

I, process 0 sent my values array: 1001 1002 1003 1004 1005 1006 1007 1008
I, process 0, received: 1001 1002 1009 1010 1017 1018 1025 1026
I, process 1 sent my values array: 1009 1010 1011 1012 1013 1014 1015 1016
I, process 1, received: 1003 1004 1011 1012 1019 1020 1027 1028
I, process 2 sent my values array: 1017 1018 1019 1020 1021 1022 1023 1024
I, process 2, received: 1005 1006 1013 1014 1021 1022 1029 1030
I, process 3 sent my values array: 1025 1026 1027 1028 1029 1030 1031 1032
I, process 3, received: 1007 1008 1015 1016 1023 1024 1031 1032
```

Collective Communications

Global Reduction

- A `reduction` is an operation applied to a set of elements in order to obtain one single value. Typical examples are the sum of the elements of a vector (SUM(A(:))) or the search for the maximum value element in a vector (MAX(V(:))).
- MPI proposes high-level subroutines in order to operate reductions on data distributed on a group of processes. The result is obtained on only one process (`MPI_Reduce()`) or on all the processes (`MPI_Allreduce()` , which is in fact equivalent to an `MPI_Reduce()` followed by an `MPI_Bcast()`).
- If several elements are implied by process, the reduction function is applied to each one of them (for instance to each element of a vector).

Collective Communications

Global Reduction

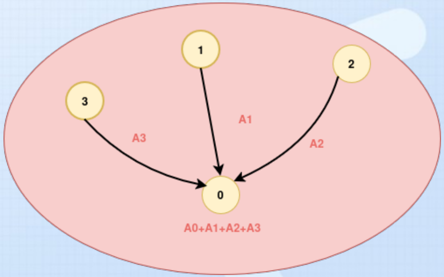


Figure: Distributed reduction (sum)

Collective Communications

Global Reduction: Operations

Name	Operation
`MPI_SUM`	Sum of elements
`MPI_PROD`	Product of elements
`MPI_MAX`	Maximum of elements
`MPI_MIN`	Minimum of elements
`MPI_MAXLOC`	Maximum of elements and location
`MPI_MINLOC`	Minimum of elements and location
`MPI_LAND`	Logical AND
`MPI_LOR`	Logical OR
`MPI_LXOR`	Logical exclusive OR

Table: Global Reduction available operations

Collective Communications

Global Reduction: MPI_Reduce()

```
int MPI_Reduce(const void*sendbuf,void *recvbuf,int count, MPI_Datatype datatype,MPI_Op op,int ↩
      root,
MPI_Comm comm)
```

- Distributed reduction of count elements of type datatype, starting at position sendbuf, with the operation op from each process of the communicator comm,
- Return the result at position recvbuf in the process root

Collective Communications

Global Reduction: MPI_Reduce() Full example

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

int main(int argc, char *argv[]) {
    int rank, nb_procs, value, sum;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
    value = 1000;
    else
    value = rank;

    MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
    printf("I, process 0, have the global sum value %d\n", sum);

    MPI_Finalize();
}
```

Global Reduction: MPI_Reduce() Full example

```
mpirun -n 8 reduce

I, process 0, have the global sum value 1028
```

Collective Communications

Global Reduction: MPI_Allreduce()

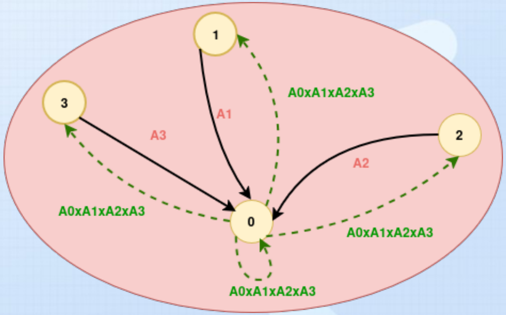


Figure: Distributed reduction (product) with distribution of the result

Collective Communications

Global Reduction: MPI_Allreduce()

```
int MPI_Allreduce(const void *sendbuf,void *recvbuf,int count, MPI_Datatype datatype,MPI_Op op,↩
     MPI_Comm comm)
```

- Distributed reduction of count elements of type `datatype` starting at position `sendbuf`, with the operation `op` from each process of the communicator `comm`,
- Write the result at position `recvbuf` for all the processes of the communicator `comm`.

Collective Communications

Global Reduction: MPI_Allreduce() Full example

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

int main(int argc, char *argv[]) {
   int rank, nb_procs, value, product;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nb_procs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   if (rank == 0)
   value = 10;
   else
   value = rank;

   MPI_Allreduce(&value, &product, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);

   printf("I, process %d, received the value of the global product %d\n",
   rank, product);

   MPI_Finalize();
}
```

Collective Communications

Global Reduction: MPI_Allreduce() Full example

```
mpirun -n 8 allreduce

I, process 7, received the value of the global product 50400
I, process 0, received the value of the global product 50400
I, process 1, received the value of the global product 50400
I, process 2, received the value of the global product 50400
I, process 3, received the value of the global product 50400
I, process 4, received the value of the global product 50400
I, process 5, received the value of the global product 50400
I, process 6, received the value of the global product 50400
```

Collective Communications

Additions

- `MPI_Scan()` allows making partial reductions by considering, for each process, the previous processes of the communicator and itself.
- `MPI_Exscan()` is the exclusive version of `MPI_Scan()`, which is inclusive.
- The `MPI_Op_create()` and `MPI_Op_free()` subroutines allow personal reduction operations.
- For each reduction operation, the keyword `MPI_IN_PLACE` can be used to keep the result in the same place as the sending buffer (but only for the rank(s) that will receive results). Example:

```
MPI_Allreduce(MPI_IN_PLACE,sendrecvbuf,...);
```

Collective Communications

Additions

- Similarly to what we have seen for `MPI_Gatherv()` with respect to `MPI_Gather()`, the following subroutines extend their respective counterparts to handle cases where processes have different numbers of elements to transmit or gather:
  - `MPI_Scatterv()` extends `MPI_Scatter()`.
  - `MPI_Allgatherv()` extends `MPI_Allgather()`.
  - `MPI_Alltoallv()` extends `MPI_Alltoall()`.
- `MPI_Alltoallw()` is the version of `MPI_Alltoallv()` that enables dealing with heterogeneous elements. It does so by expressing the displacements in bytes instead of elements.

Communication Modes

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard Send	`MPI_Send()`	`MPI_Isend()`
Synchronous Send	`MPI_Ssend()`	`MPI_Issend()`
Buffered send	`MPI_Bsend()`	`MPI_Ibsend()`
Receive	`MPI_Recv()`	`MPI_Irecv()`

Table: Point-to-Point Send Modes

Communication Modes

General concepts

Blocking call

- A call is blocking if the memory space used for the communication can be reused immediately after the exit of the call
- The data sent can be modified after the call.
- The data received can be read after the call.

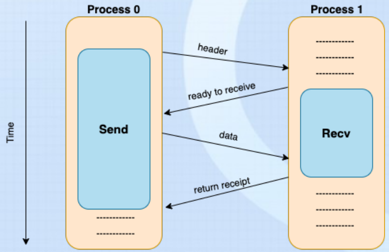
Communication Modes

Synchronous sends

- A synchronous send involves a synchronization between the involved processes. A send cannot start until its receive is posted. There can be no communication before the two processes are ready to communicate.

Rendezvous Protocol

- The rendezvous protocol is generally the protocol used for synchronous sends (implementation-dependent). The return receipt is optional.



Communication Modes

Interface of : MPI_SSEND()

```
int MPI_Ssend(const void* values,int count, MPI_Datatype msgtype, int dest,int tag, MPI_Comm comm↩
    )
```

- Advantages of synchronous mode
  - Low resource consumption (no buffer)
  - Rapid if the receiver is ready (no copying in a buffer)
  - Knowledge of receipt through synchronization
- Disadvantages of synchronous mode
  - Waiting time if the receiver is not there/not ready
  - Risk of deadlocks

Communication Modes

Deadlock example

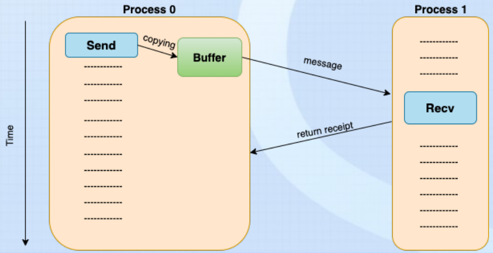
In this example, there is a deadlock because we are in synchronous mode. The two processes are blocked on the `MPI_Ssend()` call because they are waiting for the `MPI_RECV()` of the other process. However, the `MPI_Recv()` call can only be made after the unblocking of the `MPI_Ssend()` call.

```c
int main(int argc, char *argv[]) {
    int rank, num_proc, tmp, value;
    int tag = 110;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* On suppose qu'il y a exactement 2 processus */
    num_proc = (rank + 1) % 2;

    tmp = rank + 1000;
    MPI_Ssend(&tmp, 1, MPI_INT, num_proc, tag, MPI_COMM_WORLD);
    MPI_Recv(&value, 1, MPI_INT, num_proc, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    printf("Moi, processus %d, j'ai reçu %d du processus %d\n", rank, value, num_proc);

    MPI_Finalize();
}
```

Communication Modes

Buffered sends

- A buffered send implies the copying of data into an intermediate memory space. There is then no coupling between the two processes of communication. Therefore, the return of this type of send does not mean that the receive has occurred.
- Protocol with user buffer on the sender side
  In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.



Communication Modes

Interface of : MPI_Bsend()

- The buffers have to be managed manually (with calls to `MPI_Buffer_attach()` and `MPI_Buffer_detach()`). Message header size needs to be taken into account when allocating buffers (by adding the constant `MPI_Bsend_overhead()` for each message occurrence).
- Interfaces

```
int MPI_Buffer_attach(void *buf, int typesize)
int MPI_Buffer_detach(void *buf, int typesize)
int MPI_Bsend(const void *values,int count, MPI_Datatype msgtype, int dest, int tag, MPI_Comm ↩
    comm)
```

Communication Modes

No Deadlock example

In the following example, we don't have a deadlock because we are in buffered mode. After the copy is made in the buffer, the `MPI_Bsend()` call returns and then the `MPI_Recv()` call is made.

```c
int main(int argc, char *argv[]) {
    int rank, num_proc, tmp, value, bufsize, overhead, typesize;
    int tag = 110, nb_elt = 1, nb_msg = 1;
    int *buffer;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Type_size(MPI_INT, &typesize);

    /* Convert MPI_BSEND_OVERHEAD size (bytes) to integer number */
    overhead = (int)(1 + (MPI_BSEND_OVERHEAD * 1.0) / typesize);
    buffer = (int *)malloc(nb_msg * (nb_elt + overhead) * sizeof(int));
    bufsize = typesize * nb_msg * (nb_elt + overhead);
    MPI_Buffer_attach(buffer, bufsize);

    /* We assume to have exactly 2 processes */
    num_proc = (rank + 1) % 2;
    tmp = rank + 1000;
```

Communication Modes

No Deadlock example (continuation of the code)

```c
    MPI_Bsend(&tmp, nb_elt, MPI_INT, num_proc, tag, MPI_COMM_WORLD);
    MPI_Recv(&value, nb_elt, MPI_INT, num_proc, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);


    printf("I, process %d received %d from process %d\n", rank, value, num_proc);

    MPI_Buffer_detach(&buffer, &bufsize);
    MPI_Finalize();
}
```

```
mpirun -n 2 bsend

I, process 0 received 1001 from process 1
I, process 1 received 1000 from process 0
```

Communication Modes

Interface of : MPI_Send()

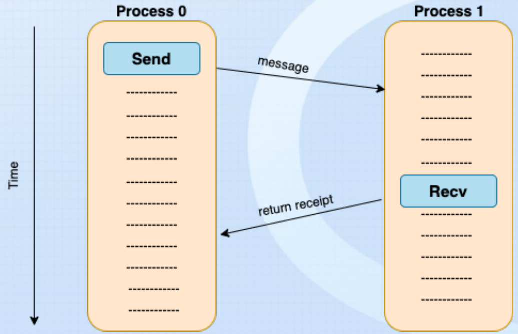
- Standard sends
  A standard send is made by calling the `MPI_Send()` subroutine. In most implementations, the mode is buffered (eager) for small messages but is synchronous for larger messages.
- Interfaces

```
int MPI_Send(const void *values, int count, MPI_Datatype msgtype, int dest, int tag, MPI_Comm ↩
    comm)
```

Communication Modes

The eager protocol

- The eager protocol is often used for standard sends of small-size messages.
- It can also be used for sends with `MPI_Bsend()` for small messages (implementation-dependent) and by bypassing the user buffer on the sender side.



Communication Modes

Number of received elements

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

- In `MPI_Recv()` call, the `count` argument in the standard is the number of elements in the buffer `buf`.
- This number must be greater than the number of elements to be received.
- When it is possible, for increased clarity, it is adviced to put the number of elements to be received.
- We can obtain the number of elements received with `MPI_Get_count()` and the `msgstatus` argument returned by the `MPI_Recv()` call

```
int MPI_Get_count(MPI_Status *msgstatus,MPI_Datatype msgtype,int *count)
```

Communication Modes

Number of received elements `MPI_Probe` allows incoming messages to be checked for, without actually receiving them.

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

A common use of `MPI_Probe` is to allocate space for a message before receiving it.

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
MPI_Get_count(&status, MPI_INT, &msgsize);
buf = (int*) malloc(msgsize*sizeof(int));
MPI_Recv(buf, msgsize, MPI_INT, status.MPI_SOURCE,
status.MPI_TAG, comm, MPI_STATUS_IGNORE);
```

Communication Modes

The overlap of communications by computations:

- A method which allows executing communication operations in the background while the program continues to operate.
- If the hardware and software architecture allows it, it is possible to hide all or part of the communications costs.
- The computation-communication overlap can be seen as an additional level of parallelism.

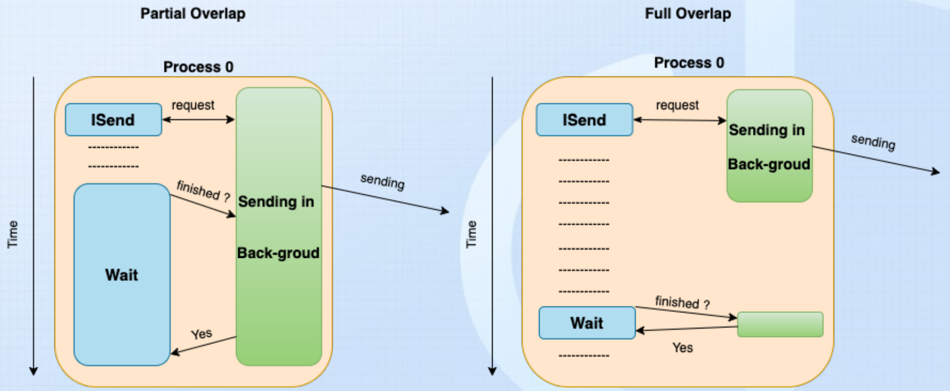
MPI Approach:

- This approach is used in MPI by using nonblocking subroutines: `MPI_Isend()`, `MPI_Irecv()` and `MPI_Wait()`

Nonblocking Communication:

- A nonblocking call returns very quickly but does not authorize the immediate re-use of the memory space which was used in the communication.
- It is necessary to make sure that the communication is fully completed (with `MPI_Wait()`, for example) before using it again.

Communication Modes



Communication Modes

Interface of : MPI_Isend(), MPI_Irecv()

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

int MPI_Issend(const void*values, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm ↩
      comm, MPI_Request *req)

int MPI_Ibsend(const void*values, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm ↩
      comm, MPI_Request *req)
```

```
int MPI_Irecv(void *values, int count, MPI_Datatype msgtype, int source, int tag, MPI_Comm comm, ↩
      MPI_Request *req)
```

Communication Modes

Interface of : MPI_Wait()

`MPI_WAIT()` wait for the end of a communication, `MPI_TEST()` is the nonblocking version.

```c
int MPI_Wait(MPI_Request *req, MPI_Status *statut)
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *statut)
```

Communication Modes

Interface of : MPI_Waitall()

`MPI_Waitall()` (`MPI_Testall()`) await the end of all communications.

```
int MPI_Waitall(int count, MPI_Request reqs[],MPI_Status statuts[])
int MPI_Testall(int count, MPI_Request reqs[],int *flag, MPI_Status statuts[])
```

Communication Modes

Interface of : MPI_Waitall()

`MPI_Waitany()` wait for the end of one communication, `MPI_Testany()` is the nonblocking version.

```c
int MPI_Waitany(int count,MPI_Request reqs[],int *indice,MPI_Status *statut)
int MPI_Testany(int count,MPI_Request reqs[],int *indice,int *flag,MPI_Status *statut)
```

`MPI_Waitsome()` wait for the end of at least one communication, `MPI_Testsome()` is the nonblocking version.

```c
int MPI_Waitsome(int count,MPI_Request reqs[],int *outcount,int *indices,MPI_Status *statuses)
int MPI_Testsome(int count,MPI_Request reqs[],int *outcount,int *indices,MPI_Status *statuses)
```

Communication Modes

Request management

- After a call to a blocking wait function (`MPI_WAIT()`, `MPI_WAITALL()`,...), the request argument is set to `MPI_REQUEST_NULL`.
- The same for a nonblocking wait when the flag is set to true.
- A wait call with a `MPI_REQUEST_NULL` request does nothing.

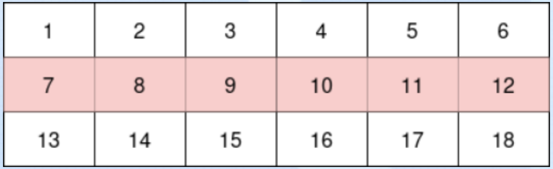
Derived datatypes

Contiguous datatypes : `MPI_Type_contiguous()`

- `MPI_Type_contiguous()` creates a data structure from a homogenous set of existing datatypes contiguous in memory.

```
int MPI_Type_contiguous(int count,MPI_Datatype old_type,MPI_Datatype *new_type)
```

```
MPI_Type_contiguous(6,MPI_INT,&new_type);
```



1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18

`MPI_Type_contiguous()` subroutine

Derived datatypes

MPI_Type_commit() and MPI_Type_free()

- Before using a new derived datatype, it is necessary to validate it with the MPI_Type_commit() subroutine.

```
int MPI_Type_commit(MPI_Datatype *new_type)
```

- The freeing of a derived datatype is made by using the MPI_Type_free() subroutine.

```
int MPI_Type_free(MPI_Datatype *new_type)
```

Derived datatypes

MPI_Type_contiguous(): **example**

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
   int rank,i,j;
   int nb_lines=6,nb_columns=5, tag=100;
   float a[nb_lines][nb_columns];

   MPI_Datatype type_line;
   MPI_Status statut;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   /* Initialization of the matrix on each process */
   for(i=0;i<nb_lines;i++)
      for(j=0;j<nb_columns;j++)
         a[i][j]=rank;

   /* Definition of the type_line datatype */
   MPI_Type_contiguous(nb_columns,MPI_FLOAT,&type_line);

   /* Validation of the type_line datatype */
   MPI_Type_commit(&type_line);
```

Derived datatypes

MPI_Type_contiguous(): **example**

```
/* Sending of the first line */
if (rank == 0) {
   MPI_Send(a,1,type_line,1,tag,MPI_COMM_WORLD);
} else {
   /* Receiving in the last line */
   MPI_Recv(&(a[nb_lines-1][0]),nb_columns,MPI_FLOAT,0,tag,
   MPI_COMM_WORLD,&statut); }

/* Print the matrix */
printf("Process %d matrix:\n", rank);
for (i = 0; i < nb_lines; i++) {
   for (j = 0; j < nb_columns; j++) {
      printf("%.1f ", a[i][j]);
   }
   printf("\n");
}
printf("\n");

/* Free the datatype */
MPI_Type_free(&type_line);
MPI_Finalize();
}
```

Derived datatypes

`MPI_Type_contiguous()`: **example**

```
mpirun -n 2 continuous
Process 1 matrix:
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
0.0 0.0 0.0 0.0 0.0

Process 0 matrix:
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
```

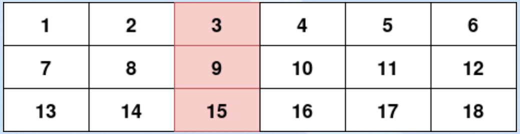
Derived datatypes

Constant stride: `MPI_Type_vector()`

- `MPI_Type_vector()` creates a data structure from a homogenous set of existing datatypes separated by a constant stride in memory. The stride is given in number of elements.

```
int MPI_Type_vector(int count,int block_length,int stride, MPI_Datatype old_type,MPI_Datatype *↩
     new_type)
```

```
MPI_Type_vector(3,1,6,MPI_FLOAT,&new_type);
```



`MPI_Type_vector()` subroutine

Derived datatypes

`MPI_Type_vector()`: example

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
   int rank,i,j;
   int nb_lines=6,nb_columns=5, tag=100;
   float a[nb_lines][nb_columns];

   MPI_Datatype type_column;
   MPI_Status statut;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   /* Initialisation of the matrix on each process */
   for(i=0;i<nb_lines;i++)
      for(j=0;j<nb_columns;j++)
         a[i][j]=rank;

   /* Definition of the datatype type_column */
   MPI_Type_vector(nb_lines,1,nb_columns,MPI_FLOAT,&type_column);

   /* Validation of the datatype type_column */
   MPI_Type_commit(&type_column);
```

Derived datatypes

MPI_Type_vector(): **example**

```
  /* Sending of the first column */
  if (rank == 0) {
     MPI_Send(&(a[1][0]),nb_lines,MPI_FLOAT,1,tag,MPI_COMM_WORLD);
  } else {
  /* Reception in the last column */
  MPI_Recv(&(a[0][nb_columns-1]),1,type_column,0,tag, MPI_COMM_WORLD,&statut); }

  /* Print the matrix */
  printf("Process %d matrix:\n", rank);
  for (i = 0; i < nb_lines; i++) {
     for (j = 0; j < nb_columns; j++) {
        printf("%.1f ", a[i][j]);
     }
     printf("\n");
  }
  printf("\n");


 /* Free the datatype type_column */
 MPI_Type_free(&type_colonne);
 MPI_Finalize();
}
```

Derived datatypes

MPI_Type_vector(): example

```
mpirun -n 2 vector

Process 1 matrix:
1.0 1.0 1.0 1.0 0.0
1.0 1.0 1.0 1.0 0.0
1.0 1.0 1.0 1.0 0.0
1.0 1.0 1.0 1.0 0.0
1.0 1.0 1.0 1.0 0.0
1.0 1.0 1.0 1.0 0.0

Process 0 matrix:
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
```

Derived datatypes

MPI_Type_vector(): send block

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
   int rank,i,j, nb_lines=6,nb_columns=5, tag=100;
   int nb_lines_block=3,nb_columns_block=2;
   float a[nb_lines][nb_columns];

   MPI_Datatype type_block;
   MPI_Status statut;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   /* Initialization of the matrix on each process */
   for(i=0;i<nb_lines;i++)
      for(j=0;j<nb_columns;j++)
         a[i][j]=rank;

   /* Creation of the datatype type_block */
   MPI_Type_vector(nb_lines_block,nb_columns_block,nb_columns, MPI_FLOAT,&type_block);

   /* Validation of the datatype type_block */
   MPI_Type_commit(&type_block);
```

Derived datatypes

MPI_Type_vector(): send block

```
    /* Sending of a block */
    if (rank == 0) {
        MPI_Send(a,1,type_block,1,tag,MPI_COMM_WORLD);
    } else {
        /* Reception of a block */
        MPI_Recv(&(a[nb_lines-3][nb_columns-2]),1,type_block,0,tag,
        MPI_COMM_WORLD,&statut); }

    /* Print the matrix */
    printf("Process %d matrix:\n", rank);
    for (i = 0; i < nb_lines; i++) {
        for (j = 0; j < nb_columns; j++) {
            printf("%.1f ", a[i][j]);
        }
        printf("\n");
    }
    printf("\n");


    /* Free the datatype type_column */
    MPI_Type_free(&type_block);
    MPI_Finalize();
}
```

Derived datatypes

`MPI_Type_vector()`: **send block**

```
mpirun -n 2 vector_block

Process 0 matrix:
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0

Process 1 matrix:
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 1.0 1.0
1.0 1.0 1.0 0.0 0.0
1.0 1.0 1.0 0.0 0.0
1.0 1.0 1.0 0.0 0.0
```

Derived datatypes

Constant stride: `MPI_Type_create_hvector()`

- `MPI_Type_create_hvector()` creates a data structure from a homogenous set of existing datatype separated by a constant stride in memory. The stride is given in bytes.
- This call is useful when the old type is no longer a base datatype (MPI_INTEGER, MPI_REAL,...) but a more complex datatype constructed by using MPI subroutines, because in this case the stride can no longer be given in number of elements.

```
int MPI_Type_create_hvector(int count,int block_length,MPI_Aint stride, MPI_Datatype old_type,↩
    MPI_Datatype *new_type)
```

Derived datatypes

Homogenous datatypes of variable strides

- `MPI_Type_indexed()`: create a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of elements.
- `MPI_Type_create_hindexed()` has the same functionality as `MPI_Type_indexed()` except that the strides separating two data blocks are given in bytes. This subroutine is useful when the old datatype is not an MPI base datatype(MPI_INTEGER, MPI_REAL, …). We cannot therefore give the stride in number of elements of the old datatype.
- For `MPI_Type_create_hindexed()`, as for `MPI_Type_create_hvector()`, use `MPI_Type_size()` or `MPI_Type_get_extent()` in order to obtain in a portable way the size of the stride in bytes.

Derived datatypes

Homogenous datatypes of variable strides: MPI_Type_indexed()

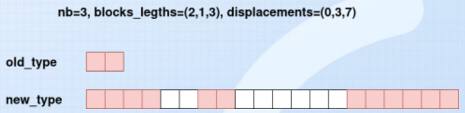


Figure: The MPI_Type_indexed() constructor

```
int MPI_Type_indexed(int nb,const int block_lengths[],const int displacements[], MPI_Datatype ↩
     old_type,MPI_Datatype *new_type)
```

Derived datatypes

Homogenous datatypes of variable strides: MPI_Type_indexed()

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

int main(int argc, char *argv[]) {
    int rank, size = 10;
    double data[size];
    int block_lengths[] = {2, 1, 3};
    int displacements[] = {0, 3, 7};
    MPI_Datatype indexed_type;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Initialize data */
    for (int i = 0; i < size; i++) {
        data[i] = rank == 0 ? i : -1;
    }

    /* Create indexed datatype */
    MPI_Type_indexed(3, block_lengths, displacements, MPI_DOUBLE, &indexed_type);
    MPI_Type_commit(&indexed_type);
```

Derived datatypes

Homogenous datatypes of variable strides: `MPI_Type_indexed()`

```c
    if (rank == 0) {
        /* Send data using indexed type */
        MPI_Send(data, 1, indexed_type, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        /* Receive data using indexed type */
        MPI_Recv(data, 1, indexed_type, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Rank %d: Received data: ", rank);
        for (int i = 0; i < size; i++) {
            printf("%.1f ", data[i]);
        }
        printf("\n");
    }

    /* Clean up */
    MPI_Type_free(&indexed_type);
    MPI_Finalize();
    return 0;
}
```

```
Rank 1: Received data: 0.0 1.0 -1.0 3.0 -1.0 -1.0 -1.0 7.0 8.0 9.0
```

Derived datatypes

Homogenous datatypes of variable strides :
`MPI_Type_create_hindexed()`

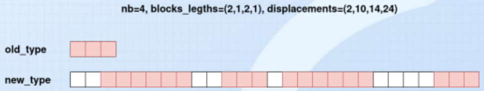


Figure: `MPI_Type_create_hindexed()` constructor

```
int MPI_Type_create_hindexed(int nb,const int block_lengths[], const MPI_Aint displacements, ↩
     MPI_Datatype old_type, MPI_Datatype *new_type)
```

Derived datatypes

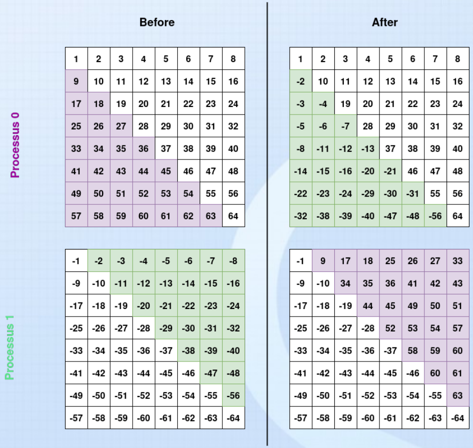
Homogenous datatypes of variable strides : `MPI_Type_indexed()`

- Example : triangular matrix

  In the following example, each of the two processes :

  1. Initializes its matrix (positive growing numbers on process 0 and negative decreasing numbers on process 1).
  2. Constructs its datatype : triangular matrix (superior for the process 0 and inferior for the process 1).
  3. Sends its triangular matrix to the other process and receives back a triangular matrix which it stores in the same place which was occupied by the sent matrix.
  4. Frees its resources.

Derived datatypes



Before – Processus 0

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

After – Processus 0

1	2	3	4	5	6	7	8
-2	10	11	12	13	14	15	16
-3	-4	19	20	21	22	23	24
-5	-6	-7	28	29	30	31	32
-8	-11	-12	-13	37	38	39	40
-14	-15	-16	-20	-21	46	47	48
-22	-23	-24	-29	-30	-31	55	56
-32	-38	-39	-40	-47	-48	-56	64

Before – Processus 1

-1	-2	-3	-4	-5	-6	-7	-8
-9	-10	-11	-12	-13	-14	-15	-16
-17	-18	-19	-20	-21	-22	-23	-24
-25	-26	-27	-28	-29	-30	-31	-32
-33	-34	-35	-36	-37	-38	-39	-40
-41	-42	-43	-44	-45	-46	-47	-48
-49	-50	-51	-52	-53	-54	-55	-56
-57	-58	-59	-60	-61	-62	-63	-64

After – Processus 1

-1	9	17	18	25	26	27	33
-9	-10	34	35	36	41	42	43
-17	-18	-19	44	45	49	50	51
-25	-26	-27	-28	52	53	54	57
-33	-34	-35	-36	-37	58	59	60
-41	-42	-43	-44	-45	-46	60	61
-49	-50	-51	-52	-53	-54	-55	63
-57	-58	-59	-60	-61	-62	-63	-64

Exchange between the two processes

Derived datatypes

Homogenous datatypes of variable strides : MPI_Type_indexed()

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char *argv[]) {
   int rank,i,j;
   int n=8, tag=100,sign=1;
   float a[n][n];

   MPI_Datatype type_triangle;
   MPI_Status statut;
   int block_lengths[n],displacements[n];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   /* Initialisation of the matrix on each process */
   if (rank == 1) sign=-1;
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         a[i][j]=sign*(1+i*n+j);
```

Derived datatypes

Homogenous datatypes of variable strides : `MPI_Type_indexed()`

```c
    /* Create type_triangle for each process */
    if (rank == 0) {
        for (i = 0; i < n; i++) {
            block_lengths[i] = i; // Upper triangle excluding diagonal
            displacements[i] = n * i; // Row offsets
        }
    } else {
        for (i = 0; i < n; i++) {
            block_lengths[i] = n - i - 1; // Lower triangle excluding diagonal
            displacements[i] = (n + 1) * i + 1; // Row offsets
        }
    }
    MPI_Type_indexed(n,block_lengths,displacements,MPI_FLOAT,&type_triangle);
    MPI_Type_commit(&type_triangle);

    /* Swap of matrix */
    MPI_Sendrecv_replace(a,1,type_triangle,(rank+1)%2,tag,(rank+1)%2,tag,MPI_COMM_WORLD,&statut);

    /* Free the triangle datatype */
    MPI_Type_free(&type_triangle);
    MPI_Finalize();
}
```

Derived datatypes

Homogenous datatypes of variable strides : `MPI_Type_indexed()`

```
mpirun -n 2 matrix_exchange

Swapped matrix on rank 0:
1.0    2.0    3.0    4.0    5.0    6.0    7.0    8.0
-2.0   10.0   11.0   12.0   13.0   14.0   15.0   16.0
-3.0   -4.0   19.0   20.0   21.0   22.0   23.0   24.0
-5.0   -6.0   -7.0   28.0   29.0   30.0   31.0   32.0
-8.0  -11.0  -12.0  -13.0   37.0   38.0   39.0   40.0
-14.0  -15.0  -16.0  -20.0  -21.0   46.0   47.0   48.0
-22.0  -23.0  -24.0  -29.0  -30.0  -31.0   55.0   56.0
-32.0  -38.0  -39.0  -40.0  -47.0  -48.0  -56.0   64.0

Swapped matrix on rank 1:
-1.0    9.0   17.0   18.0   25.0   26.0   27.0   33.0
-9.0  -10.0   34.0   35.0   36.0   41.0   42.0   43.0
-17.0  -18.0  -19.0   44.0   45.0   49.0   50.0   51.0
-25.0  -26.0  -27.0  -28.0   52.0   53.0   54.0   57.0
-33.0  -34.0  -35.0  -36.0  -37.0   58.0   59.0   60.0
-41.0  -42.0  -43.0  -44.0  -45.0  -46.0   61.0   62.0
-49.0  -50.0  -51.0  -52.0  -53.0  -54.0  -55.0   63.0
-57.0  -58.0  -59.0  -60.0  -61.0  -62.0  -63.0  -64.0
```

Derived datatypes

Size of datatype: `MPI_Type_size()`

- In order to exchange derived datatypes in communications, it is important to obtain the size of these datatypes.
- The `MPI_Type_size()` subroutine provides the size of the derived datatype in bytes.

```
int MPI_Type_size(MPI_Datatype datatype,int *typesize)
```

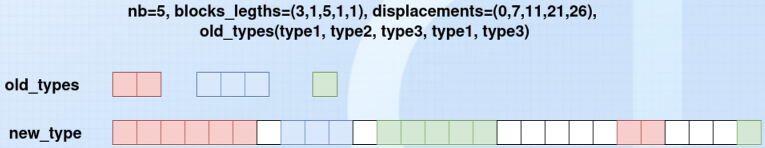
- The extent of a datatype is the memory space occupied by this datatype (in bytes). This value is used to calculate the position of the next datatype element (i.e. the stride between two successive datatype elements).

```
int MPI_Type_get_extent(MPI_Datatype datatype,MPI_Aint *lb, MPI_Aint *extent}
```

Derived datatypes

Heterogenous datatype

- `MPI_Type_create_struct()` call allows creating a set of data blocks indicating the type, the count and the displacement of each block.
- It is the most general datatype constructor. It further generalizes `MPI_Type_indexed()` by allowing a different datatype for each block.


nb=5, blocks_legths=(3,1,5,1,1), displacements=(0,7,11,21,26),
old_types(type1, type2, type3, type1, type3)
old_types
new_type

```
int MPI_Type_create_struct(int nb,const int blocks_lengths[],const MPI_Aint displacements[], ↩
    const MPI_Datatype old_types[],MPI_Datatype *new_type)
```

Derived datatypes

Compute displacements

- `MPI_Type_create_struct()` is useful for creating MPI datatypes corresponding to Fortran derived datatypes or to C structures.
- The memory alignment of heterogeneous data structures is different for each architecture and each compiler.
- Warning, you have to check the extent of the MPI datatypes obtained.
- `MPI_Get_address()` provides the address of a variable. It's equivalent of & operator in C.
- Warning, even in C, it is better to use this subroutine for portability reasons.
- It's advised to use `MPI_Aint_add()` and `MPI_Aint_diff()` to add or substract addresses

```
int MPI_Get_address(const void *variable,MPI_Aint *address_variable)
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)
```

Derived datatypes

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct Particle {
   char category[5];
   int mass;
   float coords[3];
   bool class;
};

int main(int argc,char *argv[]) {
   int rank,i;
   int n=1000,tag=100;
   int blocks_length[4];
   MPI_Datatype types[4],type_particle,temp;
   MPI_Status statut;
   MPI_Aint addresses[5],displacements[5],lb,extent;

   struct Particle p[n],temp_p[n];
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Derived datatypes

```
/* Construction of the datatype */
types[0] = MPI_CHARACTER; types[1] = MPI_INT;
types[2] = MPI_FLOAT; types[3] = MPI_LOGICAL;
blocks_length[0]=5;blocks_length[1]=1;
blocks_length[2]=3;blocks_length[3]=1;

MPI_Get_address(&(p[0]),&(addresses[0]));
MPI_Get_address(&(p[0].category),&(addresses[1]));
MPI_Get_address(&(p[0].mass),&(addresses[2]));
MPI_Get_address(&(p[0].coords),&(addresses[3]));
MPI_Get_address(&(p[0].class),&(addresses[4]));

/* Compute displacements relative to start address */
for (i=0;i<4;i++) displacements[i] = MPI_Aint_diff(addresses[i+1],addresses[0]);

MPI_Type_create_struct(4,blocks_length,displacements,types,&temp);
MPI_Get_address(&(p[1]),&(addresses[1]));
lb=0;
extent = MPI_Aint_diff(addresses[1],addresses[0]);
MPI_Type_create_resized(temp,lb,extent,&type_particle);

/* Validation of type */
MPI_Type_commit(&type_particle);
```

Derived datatypes

```
    /* Initialization of particles on each process */
    /* Send particles */
    if (rank == 0) {
        MPI_Send(&(p[0]),n,type_particle,1,tag,MPI_COMM_WORLD);
    } else {
        MPI_Recv(&(temp_p[0]),n,type_particle,0,tag,
        MPI_COMM_WORLD,&statut);
    }

    /* Free type */
    MPI_Type_free(&type_particle);
    MPI_Finalize();
}
```

Derived datatypes

Memento

Subroutines	Blocks_lengths	Strides	Old_types
`MPI_Type_Contiguous()`	constant	constant	constant
`MPI_Type_[Create_H]Vector()`	constant⋆	constant⋆	constant
`MPI_Type_[Create_H]Indexed()`	variable	variable	constant
`MPI_Type_Create_Struct()`	variable	variable	variable

(⋆) hidden parameter, equal to 1