University of Zurich[UZH]

High Performance Computing Lecture 7

Douglas Potter
Jozef Bucko
Noah Kubli

The Message Passing Interface (MPI) Library

- Introduction
- Environment
- Point-to-Point Communications
- Collective Communications
- Non-Blocking Communications
- Derived datatypes

Introduction and history

Parallel processing:

Coordinate the work between thousands to millions of processors.

MPI is a library which allows process coordination and scheduling using a message-passing paradigm.

Sequential programming model:

- The computer program is executed by only one process.
- All the variables and constants of the program are allocated in the memory of the process.
- The process is executed on a physical CPU of the node.

Message passing programming model:

- All the variables are private and reside in the local memory of each process.
- Each process can execute a different part of the program
- Variables can be exchanged between processes via a call to a message passing subroutine.

Message passing concepts

Message attributes

- The message is sent from a source process to a target process: sender address and recipient address
- The message contains a header
  - Identifier of the sending process (sender id)
  - The type of the message data (datatype)
  - The length of the message data (data length)
  - Identifier of the receiving process (receiver id)
- The message contains data

Environment

- The messages are managed and interpreted by a runtime system comparable to a telephone provider, email system or postal company.
- Message are sent to a specific address. Receiving processes must be able to classify and interpret incoming messages.
- An MPI application is a group of autonomous processes deployed on different nodes, each one executing its own code and communicating to the other processes via calls to routines in the MPI library.

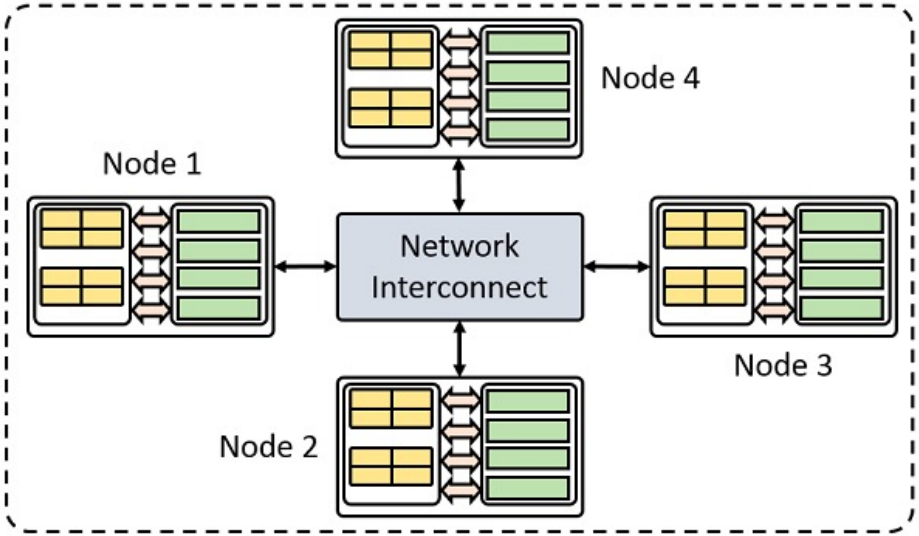
Distributed memory versus shared memory

MPI uses a distributed memory paradigm

- Data are transferred explicitly between nodes through the network

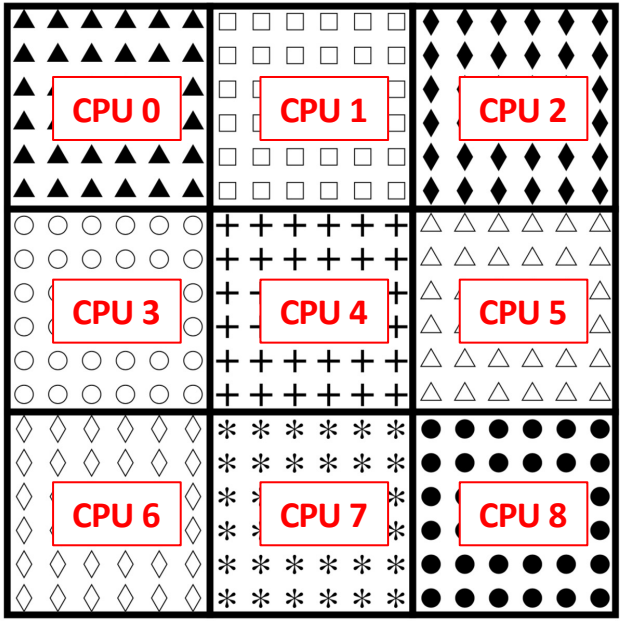
OpenMP uses a shared memory paradigm

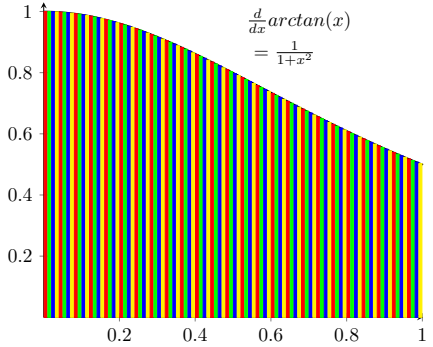
- Data are shared implicitly within the node through the Random-Access Memory.

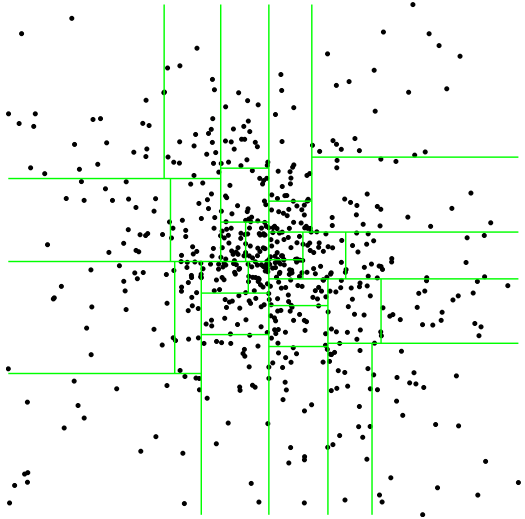


Data distribution

Data are distributed between nodes and/or cores using a domain decomposition strategy







MPI history http://mpi-forum.org

MPI 1:

- Version 1.0 (June 1994): 40 different organizations develop the MPI standard, with various subroutines defining the first MPI library
- Version 1.1 (June 1995)
- Version 1.2 (1997)
- Version 1.3 (September 2008): final version

MPI 2:

- Version 2.0 (July 1997): include new features intentionally left out of MPI 1 such as dynamical process management, one-sided communication, parallel I/O
- Version 2.1 (June 2008)
- Version 2.2 (September 2009)

MPI 3:

- Version 3.0 (September 2012): include new features left out of MPI 2 such as collective non-blocking communications, Fortran 2003 bindings, interfacing with external tools

MPI 4:

- Version 4.0 (June 2021): hybrid programming, fault tolerance, million-way parallelism

MPI implementations

Open Source libraries :

- Can be installed on almost any architecture (your laptop)
- MPICH2
- Open MPI

Vendors:

- Cray (mpich)
- Intel MPI
- Platform MPI (IBM)
- Bullx MPI

Debuggers and performance analysis tools:

- Totalview
- DDT
- Scalasca

Scientific libraries

- Scalapack
- PETSc
- FFTW

MPI Environment

MPI environment variables:

- `include mpi.h` file (MPI 1-Fortran or C/C++)
- `use mpi` module (MPI2-Fortran)

Launching the MPI environment:
`MPI_INIT()` routine

```
integer, intent(out) :: code

call MPI_INIT(code)
```

Terminating the MPI environment:
`MPI_FINALIZE()` routine

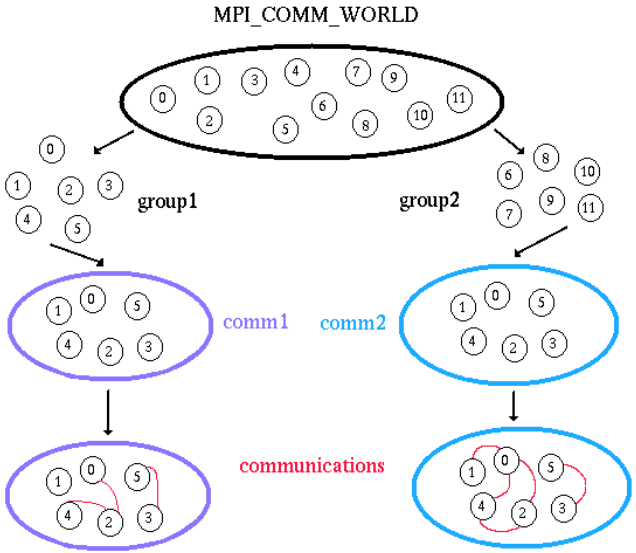
```
integer, intent(out) :: code

call MPI_FINALIZE(code)
```

C/C++ MPI routine calls

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

Communicators

All MPI communications are executed within an ensemble of processes called a *communicator*. The default communicator is identified by the MPI variable `MPI_COMM_WORLD` and includes all active processes.



Termination of a program

MPI_Abort

Terminates MPI execution environment

Synopsis

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

Input Parameters

comm
: communicator of tasks to abort

errorcode
: error code to return to invoking environment

Rank and size

MPI_Comm_size

Determines the size of the group associated with a communicator

Synopsis

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

Input Parameters

comm
communicator (handle)

Output Parameters

size
number of processes in the group of `comm` (integer)

MPI_Comm_rank

Determines the rank of the calling process in the communicator

Synopsis

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

Input Parameters

comm
communicator (handle)

Output Parameters

rank
rank of the calling process in the group of `comm` (integer)

whoami.c

```c
#include <stdio.h>
#include "mpi.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();
}
```

```
> mpirun -n 7 ./whoami
I am process 5 among 7
I am process 0 among 7
I am process 2 among 7
I am process 1 among 7
I am process 3 among 7
I am process 4 among 7
I am process 6 among 7
>
```

Point-to-point communications

General ideas

- A point-to-point communication occurs between two processes if the communicator, one called the sender and the other called the receiver.
- The sender and the receiver are identified by their rank in the communicator.
- The message header contains:
  - The rank of the sender
  - The rank of the receiver
  - The message tag
  - The communicator identifier
- The type of the exchanged data are predefined types or derived types
- For each communication, one can choose a transfer mode corresponding to different protocols.

Blocking send and receive

MPI_Send

Performs a blocking send

Synopsis

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

Input Parameters

buf
initial address of send buffer (choice)
count
number of elements in send buffer (nonnegative integer)
datatype
datatype of each send buffer element (handle)
dest
rank of destination (integer)
tag
message tag (integer)
comm
communicator (handle)

This routine send a message of **`count`** elements starting at address **`buf`** and of type **`datatype`**. This message is tagged by **`tag`** and is sent to process of rank **`dest`** within the communicator **`comm`**.

`VERY IMPORTANT:` this call is blocking. Execution remains blocked until the message can be re-written without the risk of overwriting the value being sent.

`Variants: MPI_SSEND() MPI_RSEND()`

Blocking send and receive

MPI_Recv

Blocking receive for a message

Synopsis

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

Output Parameters

buf
initial address of receive buffer (choice)
status
status object (Status)

Input Parameters

count
maximum number of elements in receive buffer (integer)
datatype
datatype of each receive buffer element (handle)
source
rank of source (integer)
tag
message tag (integer)
comm
communicator (handle)

This routine receives a message of **`count`** elements starting at address **`buf`** and of type **`datatype`**. This message must be have been tagged with **`tag`** and is coming from process of rank **`source`** within the communicator **`comm`**.
`status` receives information about the communication: source, tag, error codes…
The **`MPI_RECV`** will not work with a corresponding **`MPI_SEND`** unless the two calls have the same header (**`source, dest, tag, comm`**)

`VERY IMPORTANT:` this call is blocking. The execution remains blocked until the message is fully received and stored in **`buf`**

point2point.c

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[]) {
    int rank, size, value, tag=100;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank==2) {
        value = 1000;
        MPI_Send(&value,1,MPI_INTEGER,5,tag,MPI_COMM_WORLD);
    }
    else if (rank==5){
        MPI_Recv(&value,1,MPI_INTEGER,2,tag,MPI_COMM_WORLD,&status);
        printf("I, process 5, received %d from the process 2\n",value);
    }
    MPI_Finalize();
}
```

```
> mpirun -n 7 ./point2point
I, process 5, received 1000 from the process 2
>
```

MPI predefined datatypes

MPI Type	C Type
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`

Table 2: Predefined MPI Datatypes (C)

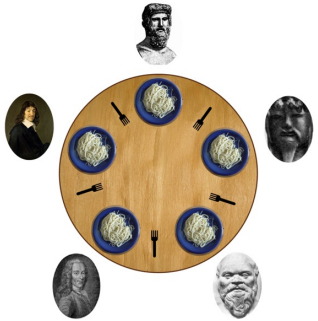
Other communication options

- On reception, the rank of the sender can be set to **`MPI_ANY_SOURCE`** and the tag of the message can be set to **`MPI_ANY_TAG.`** These are called wildcards.
- A communication with the dummy process of rank **`MPI_PROC_NULL`** will have no effect.
- **`MPI_STATUS_IGNORE`** is an MPI constant that can be used to ignore the status variable.
- A combination of **`MPI_SEND()`** and **`MPI_RECV()`** can be replaced by the variants **`MPI_SENDRECV()`** or **`MPI_SENDRCV_REPLACE()`**
- It is possible to replace the pre-defined datatypes by a user-defined derived datatype.

Deadlock

With blocking send and receive, if you are not careful, you can have a deadlock. This means the processors will wait for something that will never happen.

The classical mistake is to first call a blocking send *towards the next processor*, and then call a blocking receive *from the previous processor*.



Deadlock

With blocking send and receive, if you are not careful, you can have a deadlock. This means the processors will wait for something that will never happen. The classical mistake is to first call a blocking send *towards the next processor*, and then call a blocking receive *from the previous processor*.
MPI_Send blocks until you can reuse the buffer. MPI_Send is permitted to buffer the message either on the send or receive side, **or to wait for the matching receive**. This is implementation dependent.

deadlock.c

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[]) {
    int rank, size, value, send, other, tag=100;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    other = rank ^ 1;
    send = rank + 1000;
    MPI_Send(&send, 1,MPI_INTEGER,other,tag,MPI_COMM_WORLD);
    MPI_Recv(&value,1,MPI_INTEGER,other,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    printf("I, process %d, received %d from the process %d\n",
        rank, value, other);
    MPI_Finalize();
}
```

rank	other
0	1
1	0
2	3
3	2

Collective Communications

General concepts

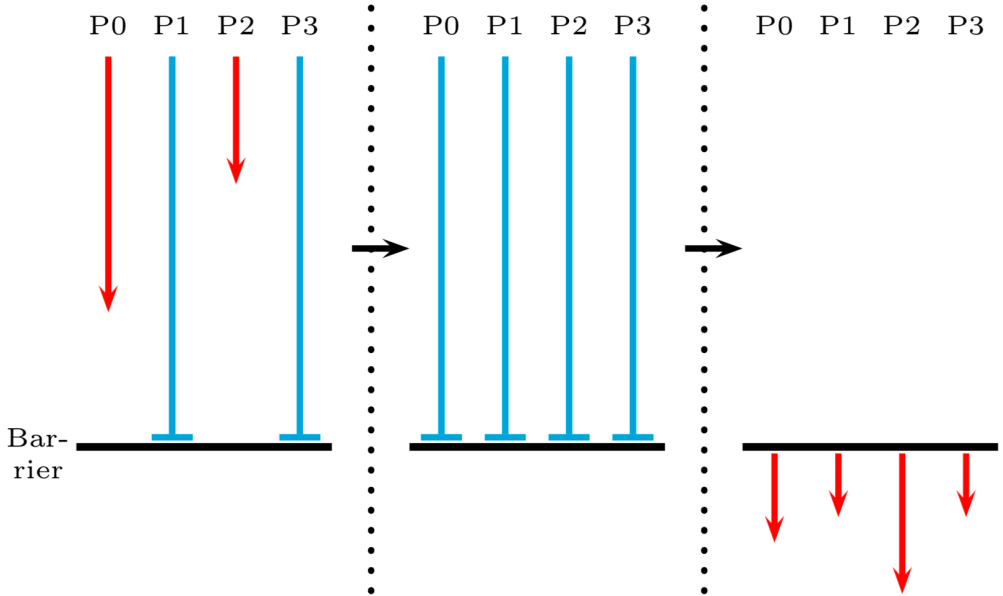
- Collective communications are making a series of point-to-point calls hidden to the user within one single subroutine
- A collective communication always involves all processes within the communicator.
- A collective communication is blocking. It is finished when all the necessary point-to-point communications are completed.
- No need to add a barrier.
- No need to specify tags.

Types of collectives

- Global synchronization **`MPI_BARRIER()`**
- Collective transfer of data
- Collective transfer plus additional operation on the data **`MPI_REDUCE()`**

Global synchronization

`MPI_BARRIER()`



Reduction operation

- A reduction is an operation applied to a set of distributed elements in order to obtain one single value.
- Operations can be for example the sum of the elements of a distributed vector or the maximum of the elements of a distributed vector.
- MPI proposes dedicated subroutines to perform these reduction on data distributed over a group of processes (within a communicator).
- **MPI_REDUCE()** computes the operation and collects the result on only one process
- **MPI_ALLREDUCE()** computes the operation and broadcast the result to all processes

Reduction operation

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

Reduction operation

MPI_Reduce

Reduces values on all processes to a single value

Synopsis

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

Input Parameters

sendbuf
address of send buffer (choice)
count
number of elements in send buffer (integer)
datatype
data type of elements of send buffer (handle)
op
reduce operation (handle)
root
rank of root process (integer)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice, significant only at `root`)

This routine perform a reduction over **`count`** elements starting at address **`sendbuf`** and of type **`datatype`**. The reduction is based on operation **`op`** for each process within the communicator **`comm`**.
This routine returns the result at address **`recvbuf`** only in process **`root.`**

rank	value
0	1000
1	1
2	2
3	3
4	4
5	5
6	6

reduce.c

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[]) {
    int rank, size, value, sum;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank==0) value = 1000;
    else value = rank;
    MPI_Reduce(&value,&sum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
    if (rank==0) printf("I, process 0, have the global sum %d\n",sum);
    MPI_Finalize();
}
```

```
> mpirun -n 7 ./reduce
I, process 0, have the global sum 1021
>
```

Reduction Operation

All-Reduction operations

University of Zurich[UZH]

MPI_Allreduce

Combines values from all processes and distributes the result back to all processes

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
count
number of elements in send buffer (integer)
datatype
data type of elements of send buffer (handle)
op
operation (handle)
comm
communicator (handle)

Output Parameters

recvbuf
starting address of receive buffer (choice)

This routine perform a reduction over `count` elements starting at address `sendbuf` and of type `datatype`. The reduction is based on operation `op` for each process within the communicator `comm`.
This routine returns the result at address `recvbuf` in all processes.

allreduce.c

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc,char *argv[]) {
    int rank, size, value, product;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if (rank==0) value = 10;
    else value = rank;
    MPI_Allreduce(&value,&product,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD);
    printf("I, process %d, have the global product %d\n",rank,product);
    MPI_Finalize();
}
```

```
> mpirun -n 7 ./allreduce
I, process 0, have the global product 7200
I, process 1, have the global product 7200
I, process 4, have the global product 7200
I, process 5, have the global product 7200
I, process 2, have the global product 7200
I, process 3, have the global product 7200
I, process 6, have the global product 7200
>
```

rank	value
0	10
1	1
2	2
3	3
4	4
5	5
6	6

All Reduction Operation

Broadcast operation

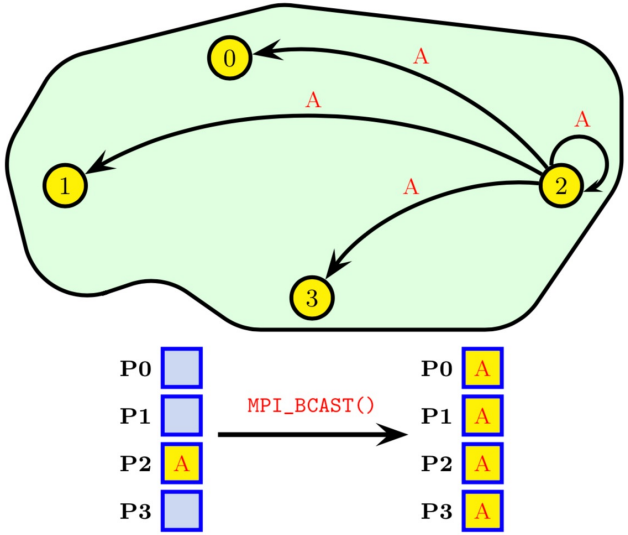


Figure 13: Global distribution : `MPI_BCAST()`

MPI_Bcast

Broadcasts a message from the process with rank "root" to all other processes of the communicator

Synopsis

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

Input/Output Parameters

buffer
starting address of buffer (choice)

Input Parameters

count
number of entries in buffer (integer)
datatype
data type of buffer (handle)
root
rank of broadcast root (integer)
comm
communicator (handle)

Send **`count`** elements starting at address **`buffer`** and of type **`datatype`**.
The broadcast is performed by and in process **`root`** towards all processes within the communicator **`comm`**.
This routine returns the result at address **`buffer`** in all processes (including **`root`**).

Broadcast

bcast.c

```c
#include <stdio.h>
#include "mpi.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_INTEGER,2,MPI_COMM_WORLD);
    printf("I, process %d, have received %d of process 2\n",rank,value);
    MPI_Finalize();
}
```

```
> mpirun -n 7 ./bcast
I, process 2, have received 1002 of process 2
I, process 4, have received 1002 of process 2
I, process 6, have received 1002 of process 2
I, process 3, have received 1002 of process 2
I, process 5, have received 1002 of process 2
I, process 0, have received 1002 of process 2
I, process 1, have received 1002 of process 2
>
```

Scatter operation

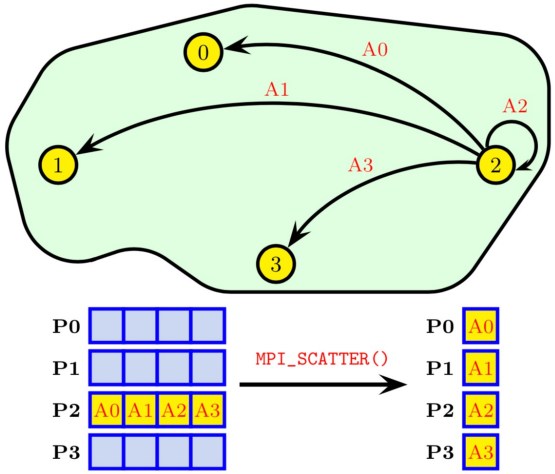


Figure 14: Selected distribution : MPI_SCATTER()

MPI_Scatter

Sends data from one process to all other processes in a communicator

Synopsis

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

Input Parameters

sendbuf
address of send buffer (choice, significant only at `root`)
sendcount
number of elements sent to each process (integer, significant only at `root`)
sendtype
data type of send buffer elements (significant only at `root`) (handle)
recvcount
number of elements in receive buffer (integer)
recvtype
data type of receive buffer elements (handle)
root
rank of sending process (integer)
comm
communicator (handle)

Send **sendcount** elements starting at address **sendbuf** and of type **sendtype**. The scatter is performed by and in process **root** towards all processes within the communicator **comm**. This routine returns a result of type **recvtype** at address **recvbuf** and size **recvcount** in all processes in communicator **comm**.

Scatter operation

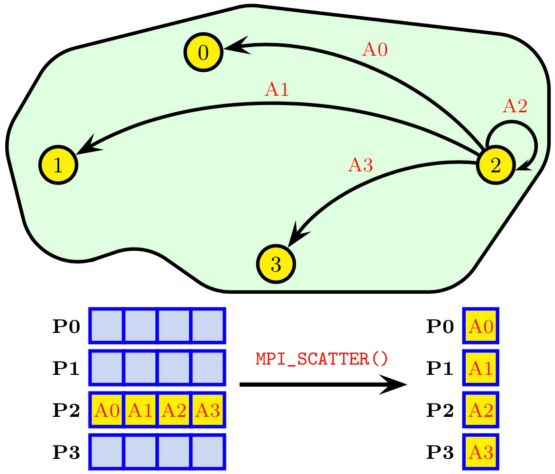


Figure 14: Selected distribution : `MPI_SCATTER()`

MPI_Scatter

Sends data from one process to all other processes in a communicator

Synopsis

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

Input Parameters

sendbuf
address of send buffer (choice, significant only at `root`)
sendcount
number of elements sent to each process (integer, significant only at `root`)
sendtype
data type of send buffer elements (significant only at `root`) (handle)
recvcount
number of elements in receive buffer (integer)
recvtype
data type of receive buffer elements (handle)
root
rank of sending process (integer)
comm
communicator (handle)

The combination of **`sendcount`** and **`sendtype`** must represent the same amount of data than the combination **`recvcount`** and **`recvtype`**. Data are scattered in chunks of equal size.

Chunk #i is always sent to processor of rank i.

Scatter

scatter.cxx

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #define N 8
4  int main(int argc,char *argv[]) {
5      int i, rank, size, block_length, *values, *data;
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,&size);
8      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9      block_length = N / size;
10     data = new int[block_length];
11     if (rank==2) {
12         values = new int[N];
13         for(i=0; i<N; ++i) values[i] = 1000 + i;
14     } else values = nullptr;
15     MPI_Scatter(values,block_length,MPI_INTEGER,data,block_length,MPI_INTEGER,2,MPI_COMM_WORLD);
16     printf("I, process %d, received %d %d of process %d\n",rank,data[0],data[1],2);
17     MPI_Finalize();
18 }
```

```
> mpirun -n 4 ./scatter
I, process 3, received 1006 1007 of process 2
I, process 1, received 1002 1003 of process 2
I, process 2, received 1004 1005 of process 2
I, process 0, received 1000 1001 of process 2
>
```

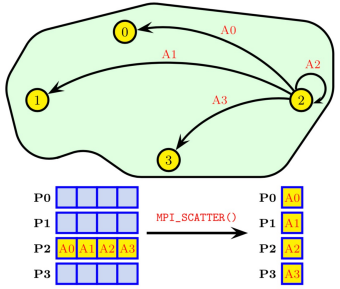


Figure 14: Selected distribution : MPI_SCATTER()

Gather operation

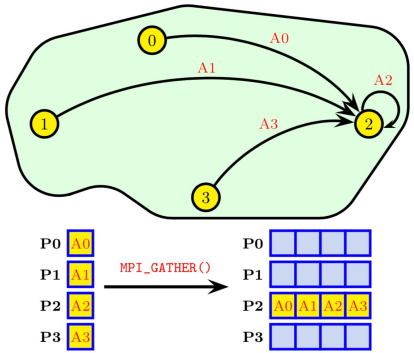


Figure 15: Collection: MPI_GATHER()

MPI_Gather

Gathers together values from a group of processes

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
sendcount
number of elements in send buffer (integer)
sendtype
data type of send buffer elements (handle)
recvcount
number of elements for any single receive (integer, significant only at root)
recvtype
data type of recv buffer elements (significant only at root) (handle)
root
rank of receiving process (integer)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice, significant only at `root`)

Send **`sendcount`** elements starting at address **`sendbuf`** and of type **`sendtype`**. The scatter is performed by and in all processes within the communicator **`comm`** towards process **`root`**. This routine collects a result of type **`recvtype`** at address **`recvbuf`** and size **`recvcount`** only in process **`root`**.

Gather operation

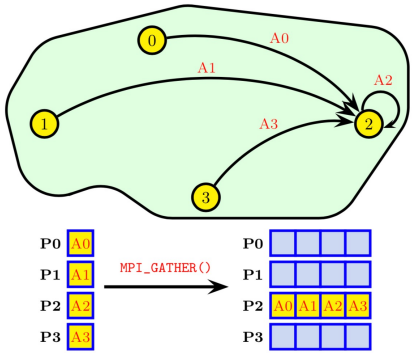


Figure 15: Collection: `MPI_GATHER()`

MPI_Gather

Gathers together values from a group of processes

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
sendcount
number of elements in send buffer (integer)
sendtype
data type of send buffer elements (handle)
recvcount
number of elements for any single receive (integer, significant only at root)
recvtype
data type of recv buffer elements (significant only at root) (handle)
root
rank of receiving process (integer)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice, significant only at `root`)

The combination of **`sendcount`** and **`sendtype`** must represent the same amount of data than the combination **`recvcount`** and **`recvtype`**.
Data is collected in process **`root`** in the same order as the other processors' ranks.

Gather

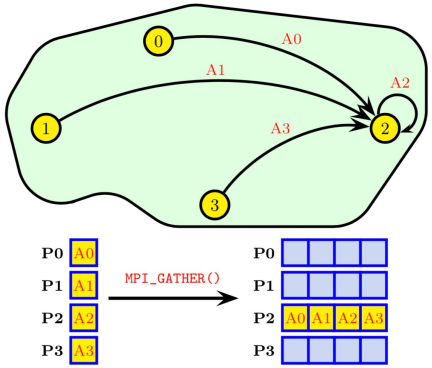


Figure 15: Collection: `MPI_GATHER()`

gather.cxx

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #define N 8
4  int main(int argc,char *argv[]) {
5      int i, rank, size, block_length, *values, data[N];
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,&size);
8      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9      block_length = N / size;
10     values = new int[block_length];
11     for(i=0; i<block_length; ++i) values[i] = 1000 + rank*block_length + i;
12     printf("I, process %d, set my values array to %d %d\n",rank,values[0],values[1]);
13     MPI_Gather(values,block_length,MPI_INTEGER,data,block_length,MPI_INTEGER,2,MPI_COMM_WORLD);
14     if (rank==2) {
15         printf("I, process 2, received");
16         for(i=0; i<N; ++i) printf(" %d",data[i]);
17         printf("\n");
18     }
19     MPI_Finalize();
20 }
```

```
> mpirun -n 4 ./gather
I, process 2, set my values array to 1004 1005
I, process 3, set my values array to 1006 1007
I, process 0, set my values array to 1000 1001
I, process 1, set my values array to 1002 1003
I, process 2, received 1000 1001 1002 1003 1004 1005 1006 1007
>
```

All-gather operation

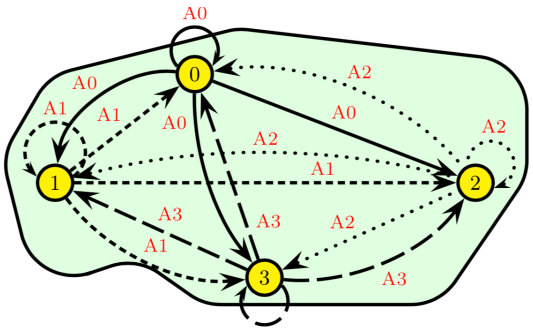


Figure 16: Gather-to-all: `MPI_ALLGATHER()`

MPI_Allgather

Gathers data from all tasks and distribute the combined data to all tasks

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
sendcount
number of elements in send buffer (integer)
sendtype
data type of send buffer elements (handle)
recvcount
number of elements received from any process (integer)
recvtype
data type of receive buffer elements (handle)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice)

It is a combination of a **`MPI_GATHER()`** followed by a **`MPI_BCAST()`**.

Same rules apply.

All Gather

allgather.cxx

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #define N 8
4  int main(int argc,char *argv[]) {
5      int i, rank, size, block_length, *values, data[N];
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,&size);
8      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9      block_length = N / size;
10     values = new int[block_length];
11     for(i=0; i<block_length; ++i) values[i] = 1000 + rank*block_length + i;
12     MPI_Allgather(values,block_length,MPI_INTEGER,data,block_length,MPI_INTEGER,MPI_COMM_WORLD);
13     printf("I, process %d, received",rank);
14     for(i=0; i<N; ++i) printf(" %d",data[i]);
15     printf("\n");
16     MPI_Finalize();
17 }
```

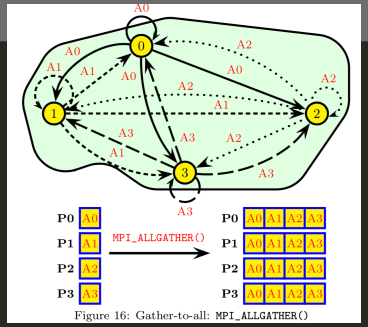


Figure 16: Gather-to-all: MPI_ALLGATHER()

```
> mpirun -n 4 ./allgather
I, process 3, received 1000 1001 1002 1003 1004 1005 1006 1007
I, process 0, received 1000 1001 1002 1003 1004 1005 1006 1007
I, process 1, received 1000 1001 1002 1003 1004 1005 1006 1007
I, process 2, received 1000 1001 1002 1003 1004 1005 1006 1007
>
```

All-to-all operation

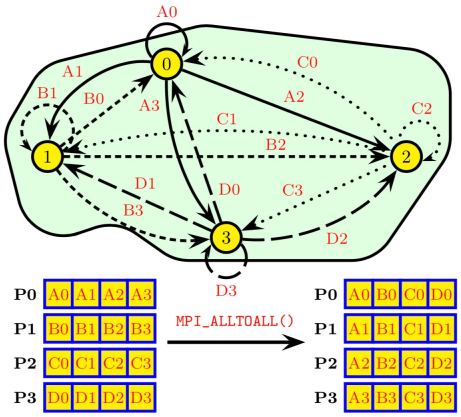


Figure 18: Collection and distribution: `MPI_ALLTOALL()`

MPI_Alltoall

Sends data from all to all processes

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
sendcount
number of elements to send to each process (integer)
sendtype
data type of send buffer elements (handle)
recvcount
number of elements received from any process (integer)
recvtype
data type of receive buffer elements (handle)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice)

It is the same as **`MPI_ALLGATHER()`** except that each process send and receives different data: Process #i sends chunk #j to process #j, which stores it in chunk #i.

Same rules apply.

Alltoall

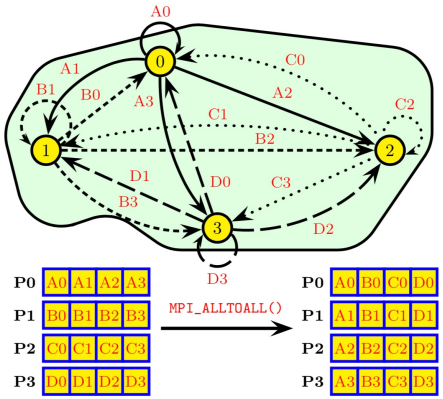


Figure 18: Collection and distribution: MPI_ALLTOALL()

alltoall.cxx

```
1  #include <stdio.h>
2  #include "mpi.h"
3  #define N 8
4  int main(int argc,char *argv[]) {
5      int i, rank, size, block_length, values[8], data[N];
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,&size);
8      MPI_Comm_rank(MPI_COMM_WORLD,&rank);
9      block_length = N / size;
10     for(i=0; i<N; ++i) values[i] = 1000 + rank*N + i;
11     printf("I, process %d, set my array to",rank);
12     for(i=0; i<N; ++i) printf(" %d",values[i]);
13     printf("\n");
14     MPI_Alltoall(values,block_length,MPI_INTEGER,data,block_length,MPI_INTEGER,MPI_COMM_WORLD);
15     printf("I, process %d, received",rank);
16     for(i=0; i<N; ++i) printf(" %d",data[i]);
17     printf("\n");
18     MPI_Finalize();
19 }
```

```
> mpirun -n 4 ./alltoall
I, process 3, set my array to 1024 1025 1026 1027 1028 1029 1030 1031
I, process 1, set my array to 1008 1009 1010 1011 1012 1013 1014 1015
I, process 0, set my array to 1000 1001 1002 1003 1004 1005 1006 1007
I, process 2, set my array to 1016 1017 1018 1019 1020 1021 1022 1023
I, process 3, received 1006 1007 1014 1015 1022 1023 1030 1031
I, process 0, received 1000 1001 1008 1009 1016 1017 1024 1025
I, process 2, received 1004 1005 1012 1013 1020 1021 1028 1029
I, process 1, received 1002 1003 1010 1011 1018 1019 1026 1027
>
```

Variable size collectives

- Same as **`MPI_GATHER()`** except that each process sends to the **`root`** process data of different size. Process #i sends array **`sendbuf`** to process **`root`**, which stores it in array **`recvbuf`**, but in chunk #i of size **`recvcounts(i)`** with an offset **`displs(i)`**.
- Otherwise, the same rules as for a classical gather apply.
- Other routines are **`MPI_SCATTERV()`**, **`MPI_ALLGATHERV()`** and **`MPI_ALLTOALLV()`**.

MPI_Gatherv

Gathers into specified locations from all processes in a group

Synopsis

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

Input Parameters

sendbuf
starting address of send buffer (choice)
sendcount
number of elements in send buffer (integer)
sendtype
data type of send buffer elements (handle)
recvcounts
integer array (of length group size) containing the number of elements that are received from each process (significant only at `root`)
displs
integer array (of length group size). Entry `i` specifies the displacement relative to recvbuf at which to place the incoming data from process `i` (significant only at root)
recvtype
data type of recv buffer elements (significant only at `root`) (handle)
root
rank of receiving process (integer)
comm
communicator (handle)

Output Parameters

recvbuf
address of receive buffer (choice, significant only at `root`)

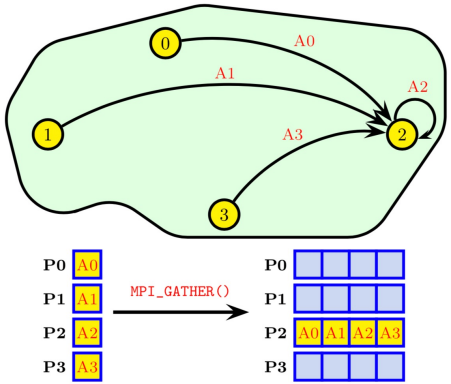


Figure 15: Collection: `MPI_GATHER()`

Non-blocking communications

General concepts

- Blocking communications can result in deadlocks
- A non-blocking communication exists the calling routine before the message has been even sent or received.
- This is possible because of (explicit or implicit) additional buffers.
- Non-blocking communications can result in memory leaks.
- Need additional routines to test the completion of the communications.
- Allow hiding communications with computations.

Non-blocking routines

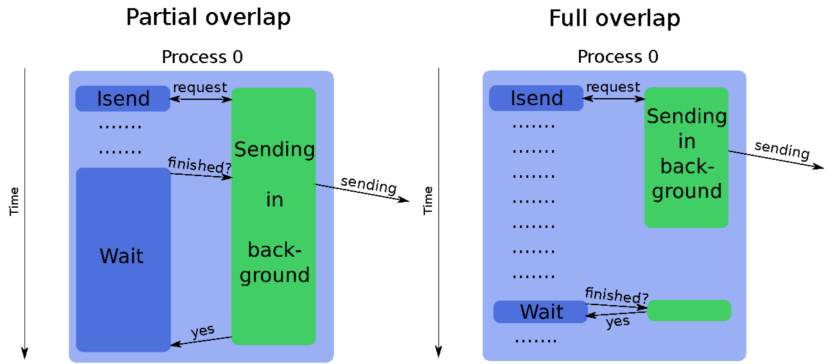
- Send and receive **`MPI_ISEND()`** and **`MPI_IRECV()`**
- Waiting until completion with **`MPI_WAIT()`**
- Testing if completion with **`MPI_TEST()`**

Non-blocking routines

Communication costs can be large. The Infiniband network latency is around microseconds, or equivalent to thousands of processor cycles. One has to add the cost of the message itself (limited by the bandwidth of the Infiniband network around 10 GB/sec).
Non-blocking routines can be used to perform calculations in parallel to the communication (additional level of parallelism). No risk of deadlock but risk of memory leak if the communication is not properly terminated.

The communication buffers however cannot be used before the proper completion of the send or receive. The user needs to test him- or herself that the communication is successful, in order to proceed with the data.

Higher algorithmic complexity.



Non-blocking Send/Recv

MPI_Irecv

Begins a nonblocking receive

Synopsis

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request * request)
```

Input Parameters

buf
initial address of receive buffer (choice)
count
number of elements in receive buffer (integer)
datatype
datatype of each receive buffer element (handle)
source
rank of source (integer)
tag
message tag (integer)
comm
communicator (handle)

Output Parameters

request
communication request (handle)

MPI_Isend

Begins a nonblocking send

Synopsis

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

Input Parameters

buf
initial address of send buffer (choice)
count
number of elements in send buffer (integer)
datatype
datatype of each send buffer element (handle)
dest
rank of destination (integer)
tag
message tag (integer)
comm
communicator (handle)

Output Parameters

request
communication request (handle)

Completing Non-blocking

MPI_Wait

Waits for an MPI request to complete

Synopsis

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

Input Parameters

request
request (handle)

Output Parameters

status
status object (Status). May be `MPI_STATUS_IGNORE`.

MPI_Test

Tests for the completion of a request

Synopsis

```
int MPI_Test(MPI_Request * request, int *flag, MPI_Status * status)
```

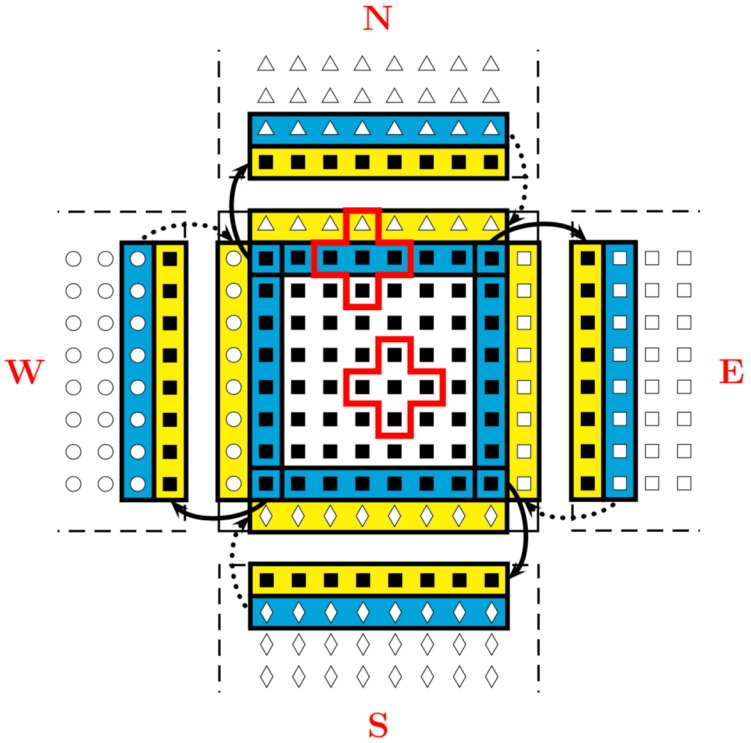
Input Parameters

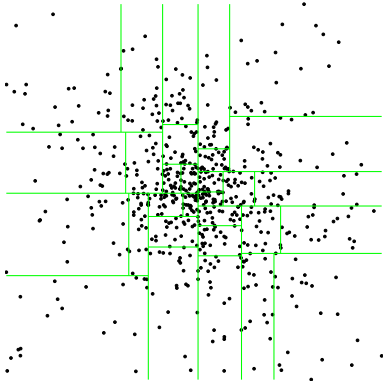
request
MPI request (handle)

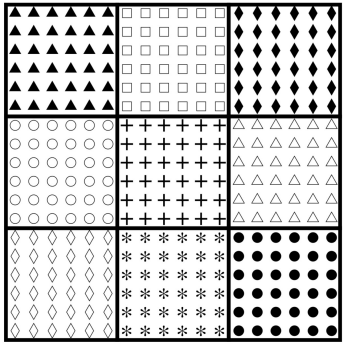
Output Parameters

flag
true if operation completed (logical)
status
status object (Status). May be `MPI_STATUS_IGNORE`.

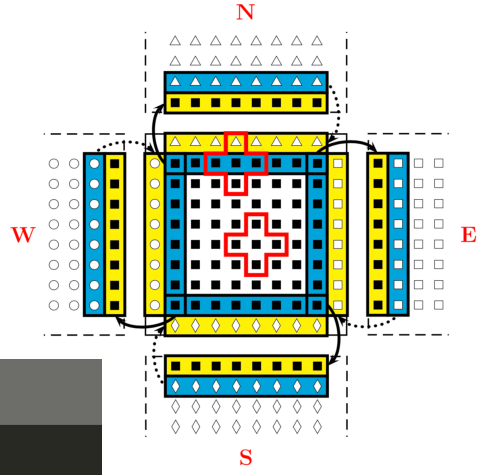
Example: managing ghost zones







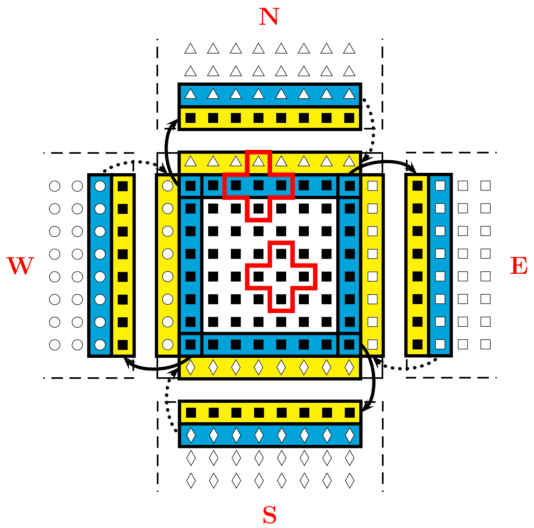
Ghost Communication



ghost.cxx

```
1  void start_communications(grid2d u) {
2      // Exchange with the north neighbor
3      MPI_Irecv(u[GTOP],   1,rowtype,neighbor(N),tag,comm,&request[0]);
4      MPI_Isend(u[TOP],    1,rowtype,neighbor(N),tag,comm,&request[1]);
5      // Exchange with the south neighbor
6      MPI_Irecv(u[GBOTTOM],1,rowtype,neighbor(S),tag,comm,&request[2]);
7      MPI_Isend(u[BOTTOM], 1,rowtype,neighbor(S),tag,comm,&request[3]);
8      // Exchange with the west neighbor
9      MPI_Irecv(u[GLEFT],  1,coltype,neighbor(W),tag,comm,&request[4]);
10     MPI_Isend(u[LEFT],   1,coltype,neighbor(W),tag,comm,&request[5]);
11     // Exchange with the east neighbor
12     MPI_Irecv(u[GRIGHT], 1,coltype,neighbor(E),tag,comm,&request[6]);
13     MPI_Isend(u[RIGHT],  1,coltype,neighbor(E),tag,comm,&request[7]);
14 }
15 void end_communication() {
16     MPI_Waitall(8,request,status);
17 }
```

```
1   DO WHILE ((.NOT. convergence) .AND. (it < it_max))
2     it = it +1
3     u(sx:ex,sy:ey) = u_new(sx:ex,sy:ey)
4
5     ! Exchange value on the interfaces
6     CALL start_communication( u )
7
8     ! Compute u
9     CALL calcul( u, u_new, sx+1, ex-1, sy+1, ey-1)
10
11    CALL end_communication( u )
12
13    ! North
14    CALL calcul( u, u_new, sx, sx, sy, ey)
15    ! South
16    CALL calcul( u, u_new, ex, ex, sy, ey)
17    ! West
18    CALL calcul( u, u_new, sx, ex, sy, sy)
19    ! East
20    CALL calcul( u, u_new, sx, ex, ey, ey)
21
22    ! Compute global error
23    diffnorm =  global_error (u, u_new)
24
25    convergence = ( diffnorm < eps )
26
27  END DO
```



Derived datatypes

General concepts

- Objected-oriented programming often relies on defining new types of data with *structures*
- **MPI_TYPE_CREATE_STRUCT()** allows to map the C or F90 structures (blocks of bytes consecutive in memory) that MPI can interpret in a machine-independent way
- Memory alignment issues are architecture and compiler dependent
- **MPI_GET_ADDRESS()** will extract from the new structure the exact address in memory of each field
- Main reason: *portability*
- Other option: convert the derived datatype structure into an array of integers or bytes and send a simple array using the Fortran intrinsic **transfer()**

Creating a new type

MPI_Type_create_struct

Create an MPI datatype from a general set of datatypes, displacements, and block sizes

Synopsis

```
int MPI_Type_create_struct(int count,
                           const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[], MPI_Datatype * newtype)
```

Input Parameters

count
: number of blocks (integer) --- also number of entries in arrays array_of_types, array_of_displacements and array_of_blocklengths

array_of_blocklengths
: number of elements in each block (array of integer)

array_of_displacements
: byte displacement of each block (array of address integer)

array_of_types
: type of elements in each block (array of handles to datatype objects)

Output Parameters

newtype
: new datatype (handle)

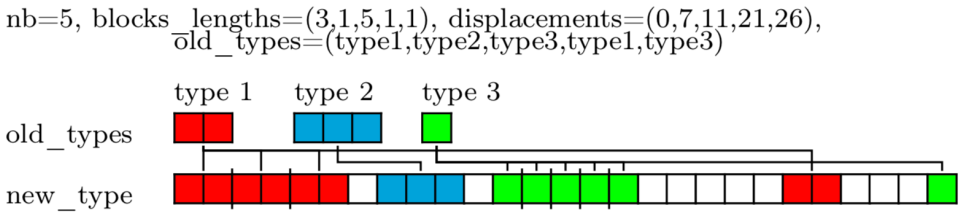


Figure 28: The MPI_TYPE_CREATE_STRUCT constructor

Managing a new type

- Must call “commit” after creating the new type
- Calling “free” releases memory

MPI_Type_commit

Commits the datatype

Synopsis

```
int MPI_Type_commit(MPI_Datatype * datatype)
```

Input Parameters

datatype
datatype (handle)

MPI_Type_free

Frees the datatype

Synopsis

```
int MPI_Type_free(MPI_Datatype * datatype)
```

Input Parameters

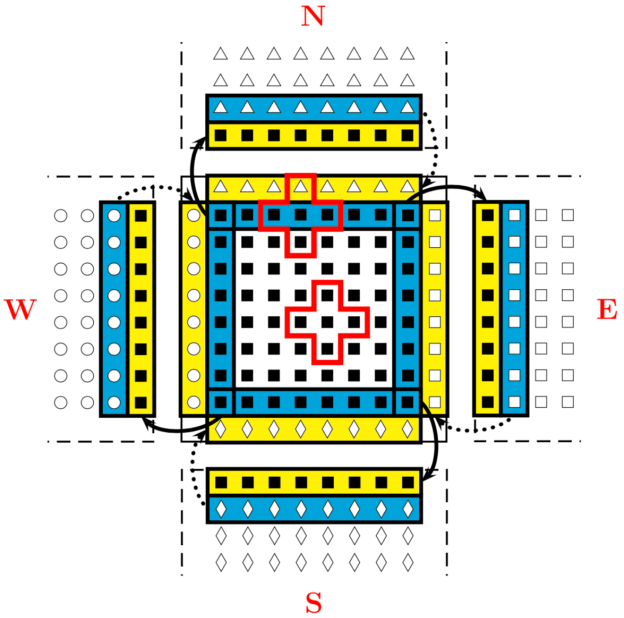
datatype
datatype that is freed (handle)

Structure Type

mpitype.cxx

```
#include "mpi.h"
struct particle {
    char category[5];
    int mass;
    float coords[3];
    bool valid;
};
int main(int argc,char *argv[]) {
    int rank, tag=99;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    int n = 1000;
    particle *particles = new particle[n];
    MPI_Datatype datatypes[] = {MPI_CHAR,MPI_INT,MPI_FLOAT,MPI_C_BOOL};
    int datacount[] = {5,1,3,1};
    MPI_Aint dataoffset[] = {offsetof(particle,category),offsetof(particle,mass),
                             offsetof(particle,coords),  offsetof(particle,valid)};
    MPI_Datatype particle_type;
    MPI_Type_create_struct(4,datacount,dataoffset,datatypes,&particle_type);
    MPI_Type_commit(&particle_type);
    // Rank 0 sends the particles to rank 1
    MPI_Status status;
    if (rank==0) MPI_Send(particles,n,particle_type,1,tag,MPI_COMM_WORLD);
    else if (rank==1) MPI_Recv(particles,n,particle_type,0,tag,MPI_COMM_WORLD,&status);
    MPI_Type_free(&particle_type);
    MPI_Finalize();
}
```

“rowtype” and “columntype”



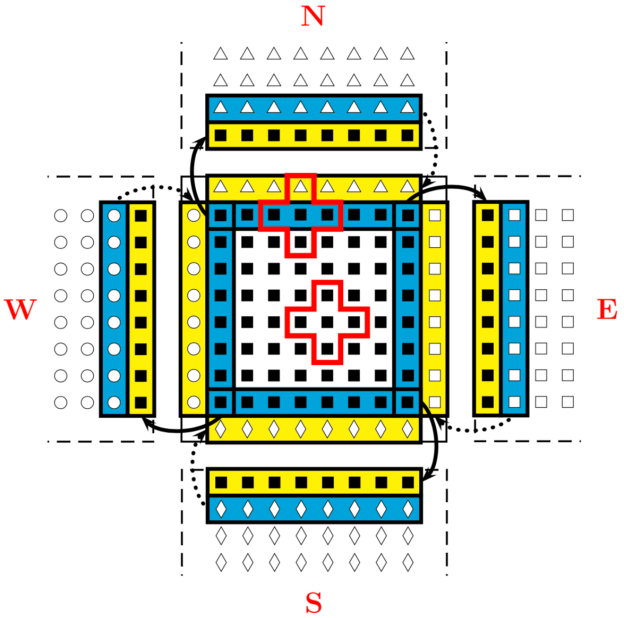
Synopsis

```
int MPI_Type_vector(int count,
                    int blocklength,
                    int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

Input Parameters

count
number of blocks (nonnegative integer)
blocklength
number of elements in each block (nonnegative integer)
stride
number of elements between start of each block (integer)
oldtype
old datatype (handle)

“rowtype” and “columntype”



Synopsis

```
int MPI_Type_create_hvector(int count,
                            int blocklength,
                            MPI_Aint stride,
                            MPI_Datatype oldtype,
                            MPI_Datatype *newtype)
```

Input Parameters

count
: number of blocks (nonnegative integer)

blocklength
: number of elements in each block (nonnegative integer)

stride
: number of bytes between start of each block (address integer)

oldtype
: old datatype (handle)

Summary

- Decide how to distribute your data
  - This is application specific
  - Goal is to keep **most** of the work local
- Determine when and what needs to be exchanged
  - This involves syncronisation
  - Beware of potential deadlocks!
  - Collectives can be safer and more convenient
    - Not always possible and result in a “barrier”
- Usage derived types if needed