MESSAGE PASSING INTERFACE - 2

Dr. Emmanuel Pilli

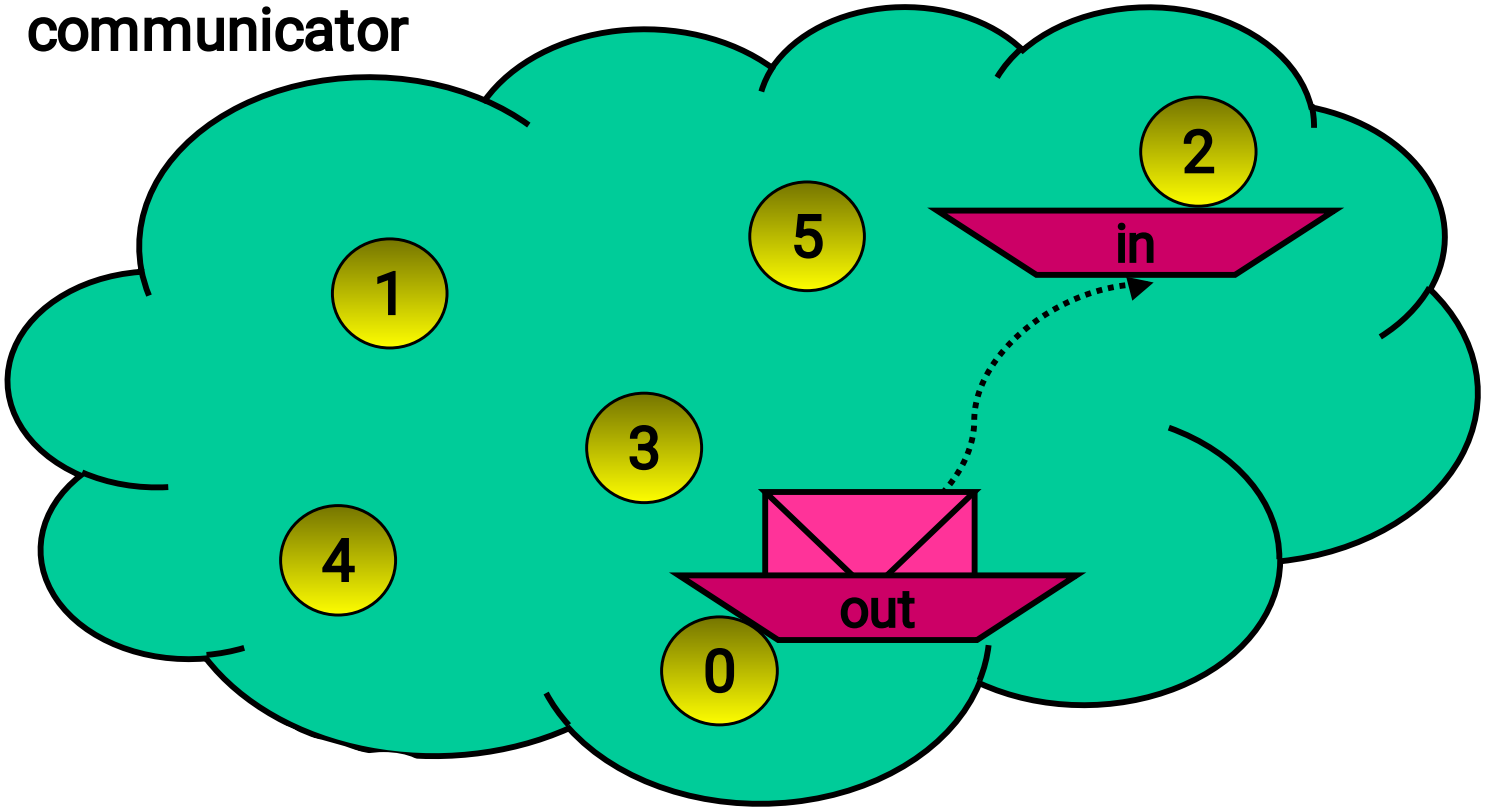
Contents

- Non-Blocking Communications
  - Send, Receive
  - Synchronous Send, Receive
- Routines
- Completion Tests
- Derived Datatypes
  - Procedure and Construction
  - Contiguous and Vector
  - Extent and Structure
  - Commit

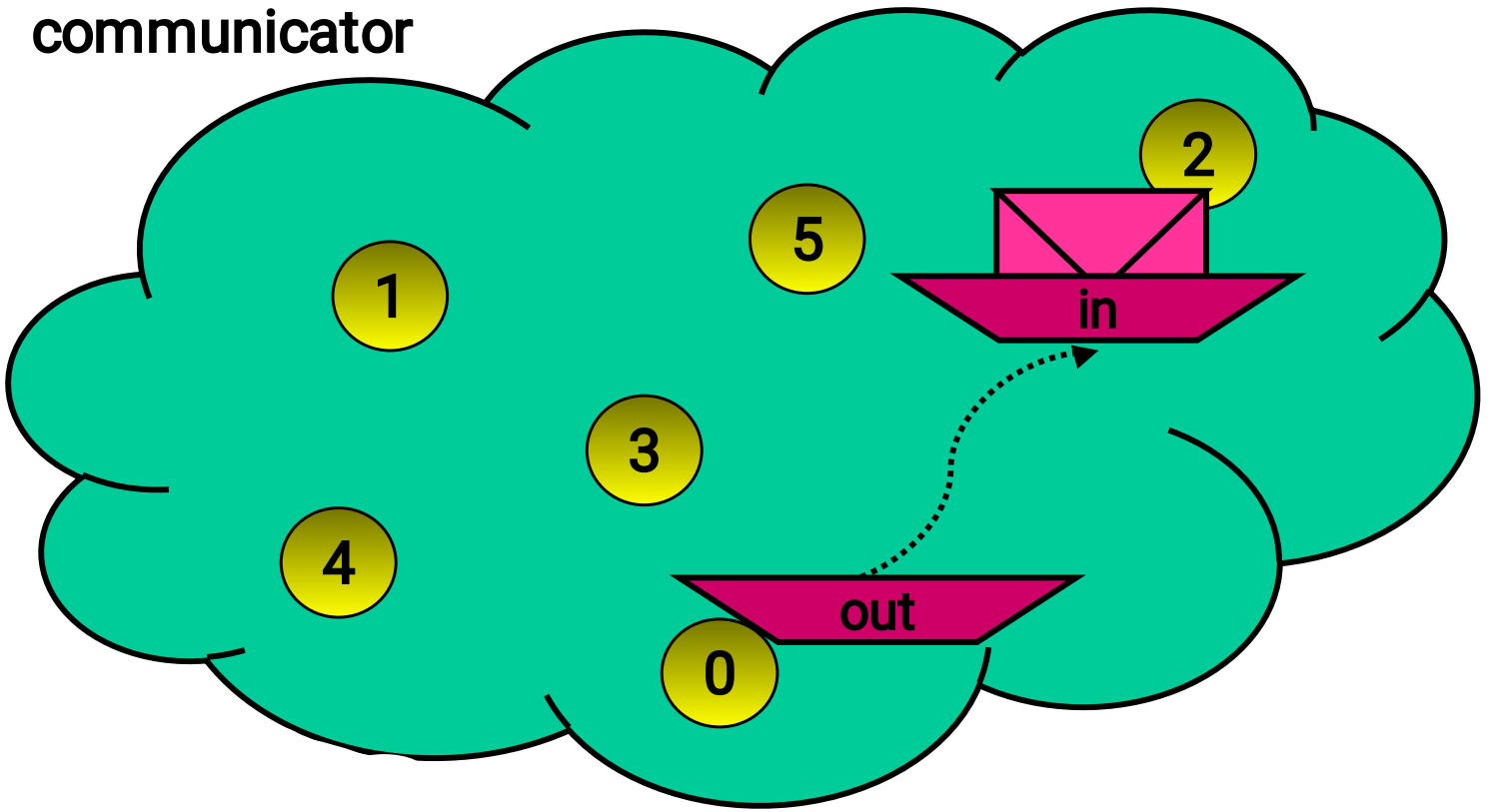
Non-Blocking Communications

- Separate communication into three phases:
  1. Initiate non-blocking communication (“post” a send or receive)
  2. Do some other work not involving the data in transfer
     - Overlap calculation and communication
     - Latency hiding
  3. Wait for non-blocking communication to complete

Non-Blocking Send



Non-Blocking Receive



Handles

- A request handle is allocated when a non-blocking communication is initiated
- The request handle is used for testing if a specific communication has completed

Datatype	Same as for blocking (MPI_Datatype or INTEGER)
Communicator	Same as for blocking (MPI_Comm or INTEGER)
Request	MPI_Request or INTEGER

Non-Blocking Synchronous Send

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

Non-Blocking Receive

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

- There is no status argument

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking
- A blocking send can be used with a non-blocking receive, and vice-versa
- Non-blocking sends can use any mode -- synchronous, buffered, standard, or ready

Routines

Non-Blocking Operation	MPI Call
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_IBSEND
Ready send	MPI_IRSEND
Receive	MPI_IRECV

Completion Tests

- **wait** and **test**

- **wait** - routine does not return until completion finished

- **test** - routine returns a TRUE or FALSE value depending on whether or not the communication has completed

Completion Tests

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

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

- Here is where status appears

Comparison

Blocking:

call **MPI_RECV (x, N, MPI_Datatype, …, status, …)**

Non-Blocking:

call **MPI_IRECV (x, N, MPI_Datatype, …, request, …)**
… do work that does not involve array x

call **MPI_WAIT (request, status)**
… do work that does involve array x

Comparison

Non-Blocking:

call **MPI_IRECV (x,N,MPI_Datatype,…,request,…)**

call **MPI_TEST (request,flag,status,…)**
do while (flag .eq. FALSE)
… work that does not involve the array x …
call **MPI_TEST (request,flag,status,…)**
end do
… do work that does involve the array x …

Multiple Communications

- Test or wait for completion of one (and only one) message

  MPI_Waitany

  MPI_Testany

- Test or wait for completion of all messages

  MPI_Waitall

  MPI_Testall

- Test or wait for completion of as many messages as possible

  MPI_Waitsome

  MPI_Testsome

Derived Datatypes

- **Derived types**
  - Constructed from existing types (basic and derived)
  - Used in MPI communication routines to transfer high-level, extensive data entities
- **Examples:**
  - Sub-arrays or “unnatural” array memory striding
  - C structures
  - Large set of general variables
- **Alternative to repeated sends of varied basic types**
  - Slow, clumsy, and error prone

Procedure

- *Construct* the new datatype using appropriate MPI routines

  MPI_Type_contiguous, MPI_Type_vector,

  MPI_Type_struct, MPI_Type_indexed,

  MPI_Type_hvector, MPI_Type_hindexed

- *Commit* the new datatype

  MPI_Type_Commit

- *Use* the new datatype in sends / receives, etc.

Construction

- **Datatype specified by its *type map***
  - Stencil laid over memory

- **Displacements are offsets (in bytes) from the starting memory address of the desired data**
  - **MPI_Type_extent** function can be used to get size (in bytes) of datatypes

Contiguous Datatype

- The simplest derived datatype consists of a number of contiguous items of the same datatype

```
int MPI_Type_contiguous (int count,
    MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

Sample Program

```
#include <stdio.h>
#include <mpi.h>
/* Run with four processes */
 void main(int argc, char *argv[]) {
    int rank;
    MPI_Status status;
    struct {
     int x;     int y;     int z;
    } point;
    MPI_Datatype ptype;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Sample Program

```
MPI_Type_contiguous(3,MPI_INT,&ptype);
  MPI_Type_commit(&ptype);
  if(rank==3){
     point.x=15; point.y=23; point.z=6;
     MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
  } else if(rank==1) {
     MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,
  &status);
     printf("P:%d received coords are (%d,%d,%d) \n",rank,
  point.x,point.y,point.z);
  }
  MPI_Finalize();
}
```

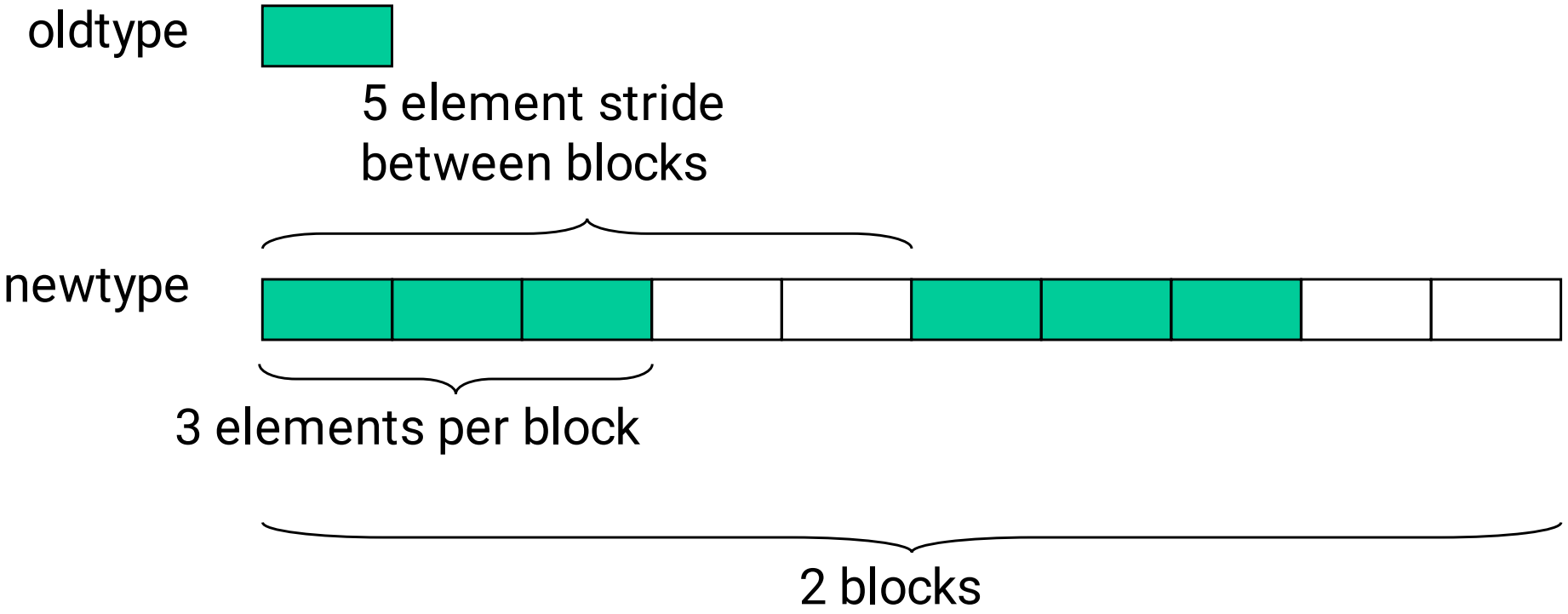
Vector Datatype

- User completely specifies memory locations defining the vector

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

- ***newtype*** has ***count*** blocks each consisting of ***blocklength*** copies of ***oldtype***
- Displacement between blocks is set by ***stride***

Vector Datatype



- count = 2, stride = 5, blocklength = 3

Sample Program

```
#include <mpi.h>
#include <math.h>
#include <stdio.h>
 void main(int argc, char *argv[]) {
    int rank,i,j;
    MPI_Status status;
    double x[4][8];
    MPI_Datatype coltype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Type_vector(4,1,8,MPI_DOUBLE,&coltype);
    MPI_Type_commit(&coltype);
```

Sample Program

```
 if(rank==3)  {
    for(i=0;i<4;++i)
     for(j=0;j<8;++j) x[i][j]=pow(10.0,i+1)+j;
 MPI_Send(&x[0][7],1,coltype,1,52,MPI_COMM_WORLD);
 }
  else if(rank==1) {
    MPI_Recv(&x[0][2],1,coltype,3,52,
                    MPI_COMM_WORLD,&status);
     for(i=0;i<4;++i)
     printf("P:%d my x[%d][2]=%1f\n",
                          rank,i,x[i][2]);
 }
   MPI_Finalize();
}
```

Extent

- Handy utility function for datatype construction
- Extent defined to be the memory span (in bytes) of a datatype

MPI_Type_extent (MPI_Datatype datatype, MPI_Aint* extent)

Structure

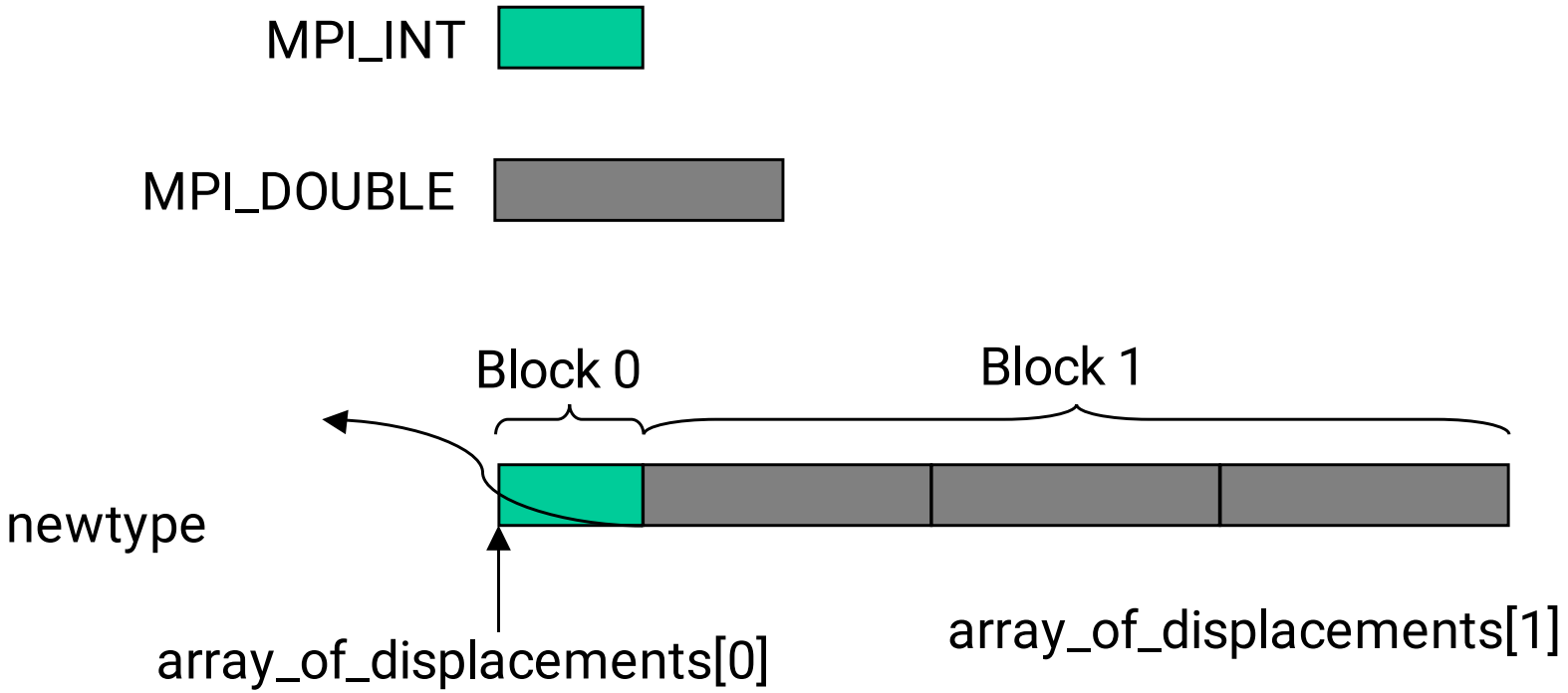
- Use for variables comprised of heterogeneous datatypes
  - C structures
- This is the most general derived data type

```
int MPI_Type_struct (int count,
  int *array_of_blocklengths,
   MPI_Aint *array_of_displacements,
   MPI_Datatype *array_of_types,
   MPI_Datatype *newtype)
```

Structure

- ***newtype*** consists of **count** blocks where the ith block is **array_of_blocklengths[i]** copies of the type **array_of_types[i].**
- The displacement of the i^{th} block (in bytes) is given by **array_of_displacements[i].**

Structure Example



- count = 2, array_of_blocklengths = {1,3}
- array_of_types = {MPI_INT, MPI_DOUBLE}
- array_of_displacements = {0, extent(MPI_INT)}

Sample Program

```
#include <stdio.h>
#include<mpi.h>
void main(int argc, char *argv[]) {
 int rank,i;
 MPI_Status status;
 struct {
    int num;
    float x;
    double data[4];
  } a;
  int blocklengths[3]={1,1,4};
  MPI_Datatype types[3] =
            {MPI_INT,MPI_FLOAT,MPI_DOUBLE};
  MPI_Aint displacements[3];
```

Sample Program

```
MPI_Datatype restype;
MPI_Aint intex,floatex;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Type_extent(MPI_INT,&intex);
MPI_Type_extent(MPI_FLOAT,&floatex);
displacements[0]= (MPI_Aint) 0;
displacemtnts[1]=intex;
displacements[2]=intex+floatex;
MPI_Type_struct(3, blocklengths,
    displacements, types, &restype);
MPI_Type_commit(&restype);
```

Sample Program

```
    if (rank==3){
      a.num=6; a.x=3.14;
       for(i=0;i 4;++i) a.data[i]=(double) i;
        MPI_Send(&a,1,restype,1,52,MPI_COMM_WORLD);
    }
    else if(rank==1)
    {
      MPI_Recv(&a,1,restype, 3, 52,
              MPI_COMM_WORLD, &status);
      printf("P:%d my a is %d %f %lf %lf %lf %lf\n",
           rank, a.num, a.x,a.data[0], a.data[1],
                        a.data[2], a.data[3]);
    }
    MPI_Finalize();
}
```

Commit

- Once a datatype has been constructed, it needs to be committed before it is used.
- This is done using **MPI_TYPE_COMMIT**

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