Tipos de datos

¿Cómo se definen y envían los datos?

- Las funciones de envío de MPI envían por defecto posiciones contiguas de memoria. Hay que tenerlo en cuenta en C:
  - En C, al definir un array bidimensional de forma dinámica, hay que asegurarse de que los elementos se almacenan de forma continua.

Ejemplo:

C:

MPI_SEND(datos, cuantos, tipo_de_dato, destino, tag, comm)

MPI_SEND(a, 5, MPI_FLOAT, 4, 1, MPI_COMM_WORLD)

- Envía los 5 números float del vector “a(5)” al proceso 4 de comm_world
- Esto sirve para datos contiguos

Tipos de datos

Envío de una fila de una matriz (caso C)

Solución 1: Enviar uno a uno los elementos de la fila.

```
double A[100][200];
for (i=0; i<200; i++)
MPI_SEND(&A[0][i], 1, MPI_DOUBLE, dest, tag, comm);
```

Solución 2: Empaquetar los datos en un vector.

```
double A[100][200];
double c[200];
for (i=0; i<200; i++)  c[i]=A[0][i];
  MPI_SEND(&c, 200, MPI_DOUBLE, dest, tag, comm);
```

Solución 3: Enviar todos los elementos de la fila con un único send

```
double A[100][200];
MPI_SEND(&A[0][0],200, MPI_DOUBLE, dest, tag, comm);
```

Tipos de datos

Como hemos mencionado anteriormente, MPI predefine sus tipos de datos primitivos. Así MPI presenta los siguientes tipos de datos:

MPI Data Types	C Data Types
MPI_CHAR	signed char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int

MPI Data Types	C Data Types
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_INT8_T MPI_INT16_T MPI_INT32_T MPI_INT64_T	int8_t int16_t int32_t int64_t
MPI_UINT8_T MPI_UINT16_T MPI_UINT32_T MPI_UINT64_T	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Tipos de datos

El usuario puede construir otros tipos de datos, datos derivados, a partir de los tipos primitivos. Los tipos de datos derivados son análogos a las estructuras de C.

Tipos de datos derivados

- Tipo contiguo
- Tipo vector
- Tipo indexado
- Tipo estructura

Tipos de datos

Tipos de datos derivados: MPI proporciona facilidades para que el usuario pueda definir sus propias estructuras de datos basadas en secuencias de los tipos de datos primitivos MPI. Tales estructuras definidas por el usuario se llaman tipos de datos derivados.
Los tipos de datos primitivos de MPI son contiguos. Los tipos de datos derivados permiten especificar datos no contiguos de una manera conveniente y tratarlos como si fueran contiguos.
MPI proporciona varios métodos para construir tipos de datos:

- Tipo contiguo
- Tipo vector
- Tipo indexado
- Tipo estructura

Al definir un nuevos tipo de dato derivado, para utilizar este nuevo tipo de dato es necesario confirmarlo con **MPI_Type_commit**. Cuando se ha acabado de utilizar, el tipo de dato derivado se libera con **MPI_Type_free**.

Tipos de datos

Tipo contiguo

Es el más simple de los tipos de datos. Define un nuevo tipo de datos, replicando una secuencia de datos contigua de un tipo existente

C

```
MPI_Type_contiguous (icount, oldtype, &newtype)
```

Tipos de datos

Tipo contiguo

Ejemplo: En C se quieren enviar M filas, a partir de la j-ésima, de una matriz A de tamaño N x N a otro proceso.

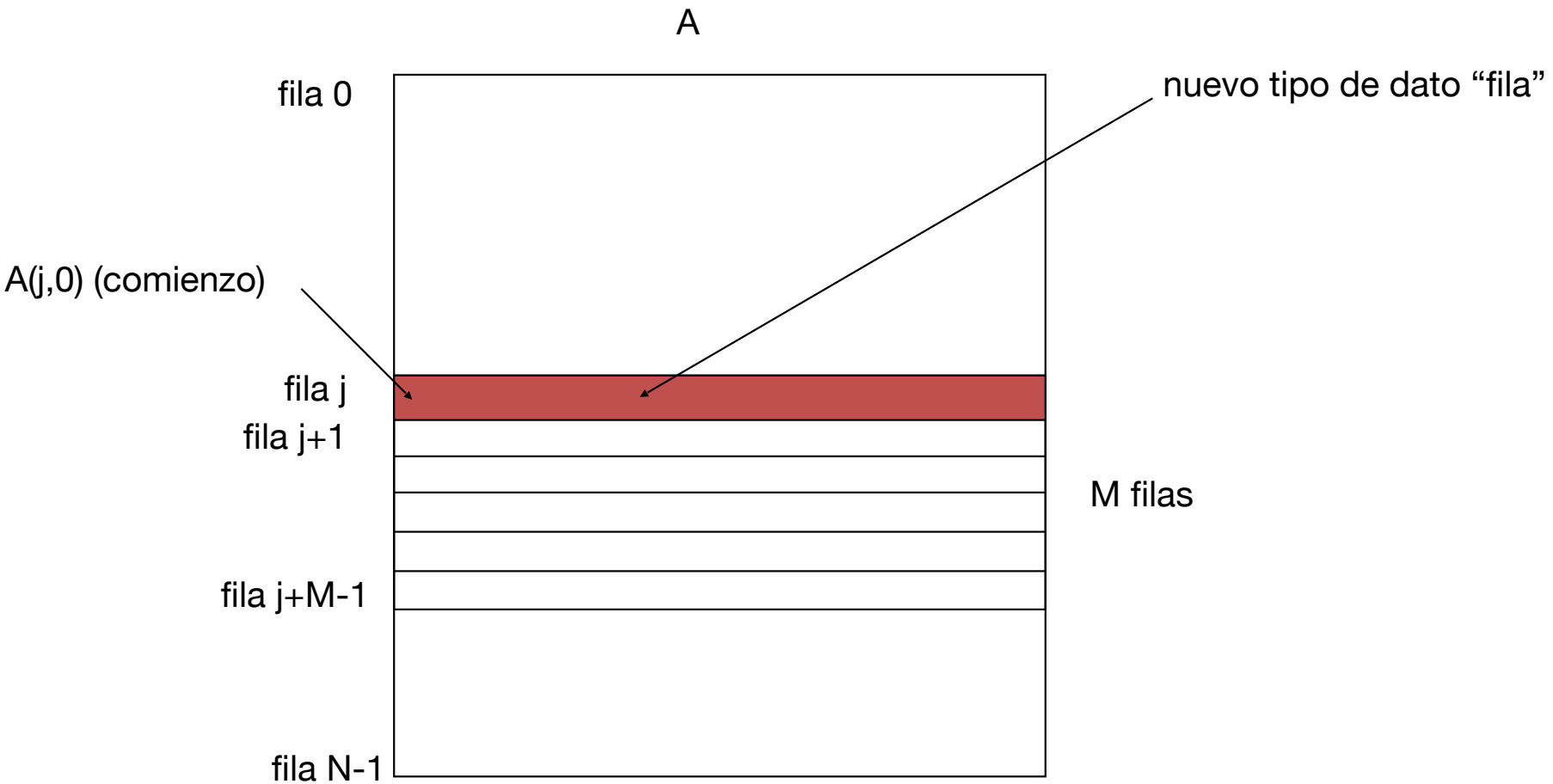
```
MPI_Datatype fila;

mpi_type_contiguous(N, MPI_FLOAT, &fila)
mpi_type_commit(&fila)
mpi_send(&A(j,0), M, fila, dest, tag, comm)
```

Numero de datos del tipo fila, i.e., M filas

Tipos de datos

MPI_Type_contiguous(N, MPI_FLOAT, &fila)

MPI_Send(&A(j,0), M, fila, dest, tag, comm)



Tipos de datos

MPI_Send (&a, count, datatype, dest, tag, comm)

Es idéntico a:

MPI_Type_contiguous(count, datatype, &newtype)
MPI_Type_commit(&newtype)
MPI_Send(&a , 1, newtype, dest, tag, comm)
MPI_Type_free(&newtype)

Tipos de datos

contiguoustype.c

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

#define size 4

int main(int argc, char *argv[])  {
int numtasks, rank, source = 0, dest, tag = 1, i;
float a[size][size] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
   13.0, 14.0, 15.0, 16.0};
float b[size];

MPI_Status stat;
MPI_Datatype rowtype;   // variable

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

// creamos tipo de datos derivado contiguous
MPI_Type_contiguous(size, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == size) {
   // task 0 envia un elemento de tipo rowtype a todos los procesos
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
      }

   // todos los procesos reciben los datos contenidos en el tipo rowtype enviados desde el proceso 0
   MPI_Recv(b, size, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
   printf("Debes ejecutar en %d procesos. Terminando.\n",size);

// Liberamos el tipo de dato cuando hemos acabado de utilizarlo
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

```
mpicc -o contiguoustype contiguoustype.c
mpirun -np 4 ./contiguoustype

rank= 1  b= 5.0 6.0 7.0 8.0
rank= 2  b= 9.0 10.0 11.0 12.0
rank= 3  b= 13.0 14.0 15.0 16.0
rank= 0  b= 1.0 2.0 3.0 4.0
```

Tipos de datos

Tipo vector

Es una pequeña generalización del anterior:
Permite definir un tipo de dato de bloques igualmente espaciados de datos contiguos.

C

```
MPI_Type_vector(cuantos, largobloque, desplazamiento,
                oldtype, &newtype)
```

Tipos de datos

C

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

Parámetros

- De entrada
  - count: Número de grupos de bloques.
  - Blocklength: Número de elementos en cada bloque.
  - stride: Número de elementos entre comienzo de cada bloque.
  - oldtype: Antiguo tipo de dato.
- De salida
  - newtype: Nuevo tipo de dato.

El nuevo tipo de dato está formado por count bloques, en el que cada bloque hay blocklength copias de oldtype. Los elementos dentro de cada bloque ocupan localizaciones contiguas, pero el desplazamiento entre cada bloque viene dado por stride.

Tipo vector

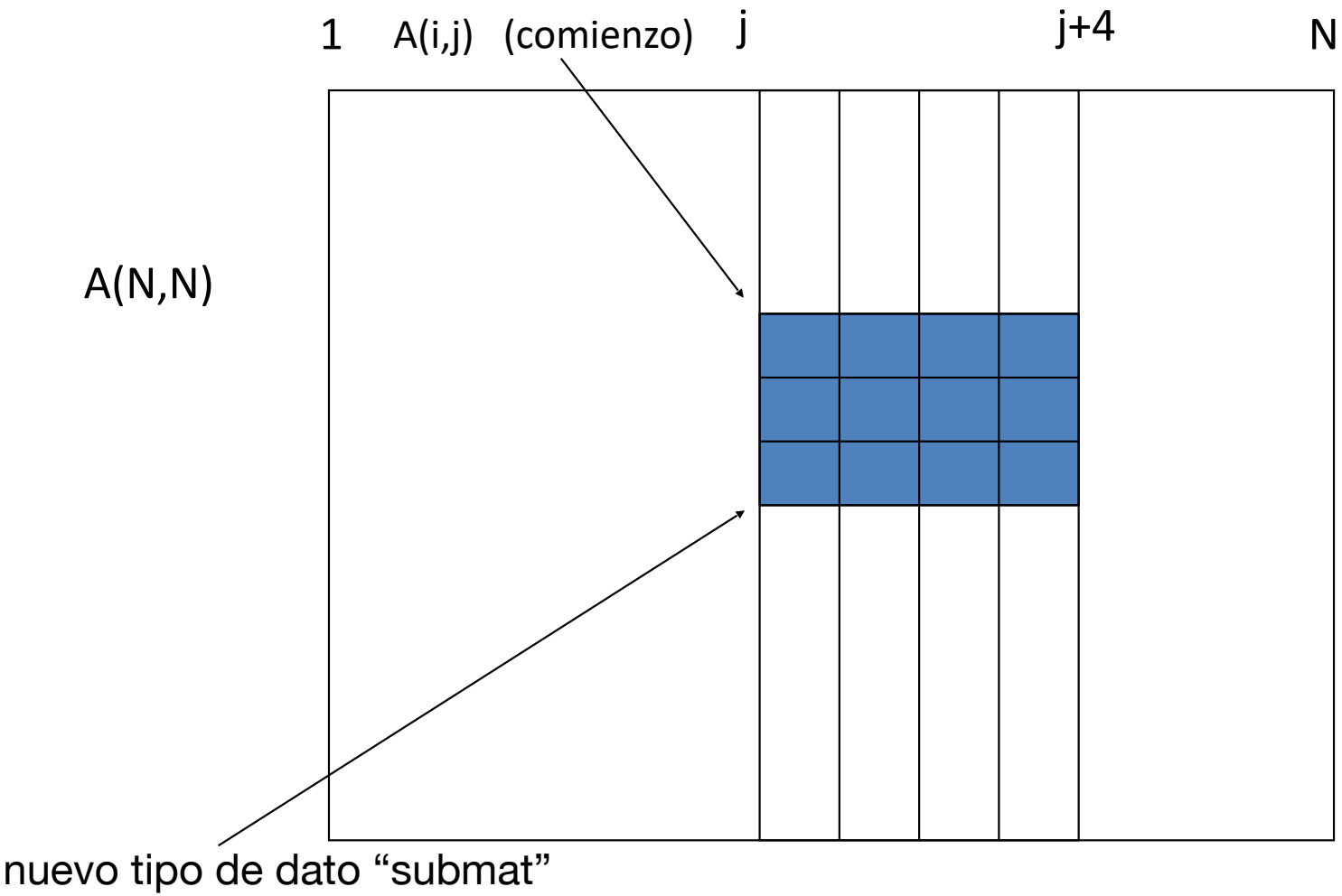
Ejemplo: definir y enviar una matrix de 3x4 embebida en una matriz A de NxN a partir del elemento A(i,j).

```
mpi_type_vector(3, 4, N, MPI_FLOAT, &submat)
mpi_type_commit(&submat)
mpi_send(&A(i,j), 1, submat, idest, itag, icomm)
```

Tipos de datos

MPI_Type_vector(3, 4, N, MPI_FLOAT, &submat)

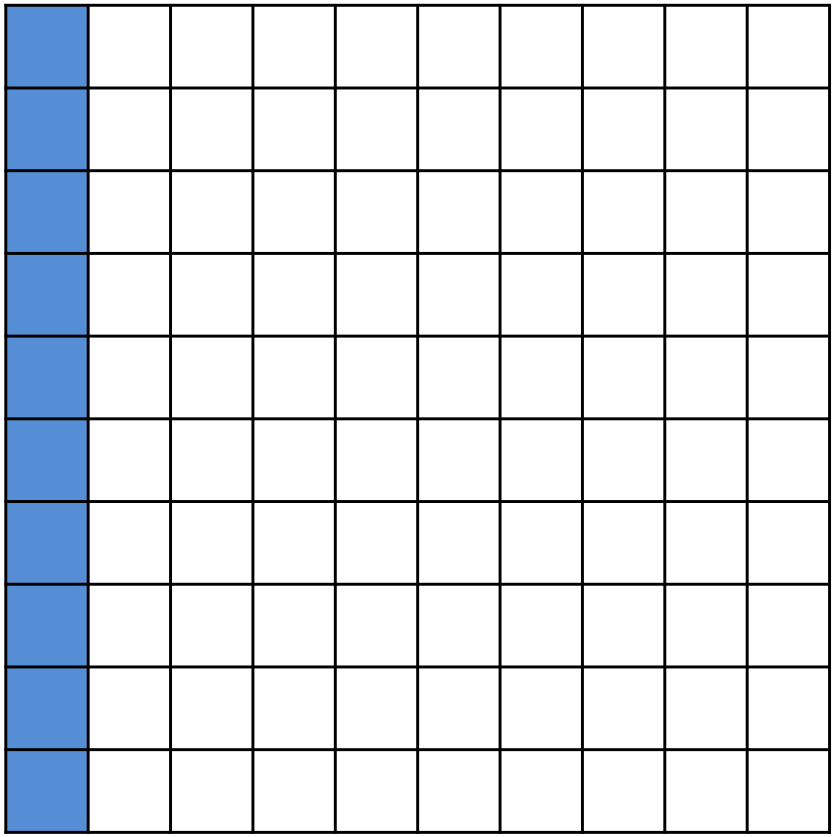
MPI_Send(A(i,j), 1, submat, idest, itag, icomm)



Tipos de datos

Ejemplo

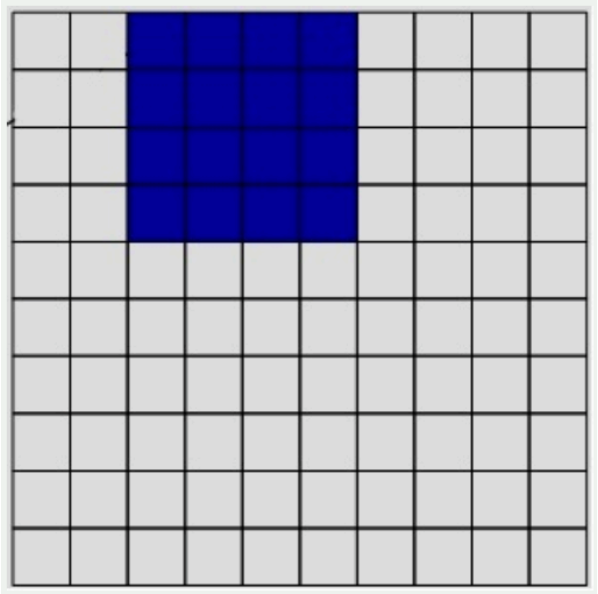
Esta llamada es útil si queremos agrupar partes de una matriz, como por ejemplo si queremos agrupar la primera columna:



- Número de bloques: 10
- Número de elementos en cada bloque: 1
- Número de elementos entre comienzo de cada bloque (stride): 10
- Antiguo tipo de dato: MPI_DOUBLE
- Nuevo tipo de dato: mitipo_columna

```
MPI_Type_vector(10, 1, 10, MPI_DOUBLE, &mitipo_columna)
```

Tipos de datos



- Número de bloques: 4
- Número de elementos en cada bloque: 4
- Número de elementos entre comienzo de cada bloque (stride): 10
- Antiguo tipo de dato: MPI_FLOAT
- Nuevo tipo de dato: mitipo_bloque

C

```
MPI_Type_vector(4, 4, 10, MPI_FLOAT, &mitipo_bloque);
```

Tipos de datos

```c
#include "mpi.h"
#include <stdio.h>
#define size 4

int  main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
float a[size][size] =
  {1.0, 2.0, 3.0, 4.0,
   5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0,
  13.0, 14.0, 15.0, 16.0};
float b[size];
MPI_Status stat;
MPI_Datatype columntype;   // variable, nuevo tipo de dato
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

// creamos tipo de datos derivado vector
MPI_Type_vector(size, 1, size, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);

if (numtasks == size) {
   // task 0 envia un elemento de tipo columntype a todos los procesos
   if (rank == 0) {
      for (i=0; i<numtasks; i++)
         MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
      }

   // todos los procesos reciben los datos columntype enviados desde el proceso 0
   MPI_Recv(b, size, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3]);
   }
else
    printf("Debes ejecutar en %d procesos. Terminando.\n",size);

    // Liberamos el tipo de dato cuando hemos acabado de utilizarlo
MPI_Type_free(&columntype);
MPI_Finalize();
}
```

```
mpicc -o vectortype vectortype.c
mpirun -np 4 ./vectortype

rank= 0  b= 1.0 5.0 9.0 13.0
rank= 1  b= 2.0 6.0 10.0 14.0
rank= 2  b= 3.0 7.0 11.0 15.0
rank= 3  b= 4.0 8.0 12.0 16.0
```

Tipos de datos

C

```
MPI_Type_commit(&datatype)
```

Parámetros

- De entrada
  - datatype: Tipo de dato.

Descripción

- Confirma el nuevo tipo de datos en el sistema. Obligatorio para todos los tipos de datos construidos por el usuario (derivados).
- Un tipo de dato es la descripción formal de un buffer de comunicación, no el contenido de tal. Después de confirmado el tipo de dato se puede reutilizar repetidamente para las comunicaciones, cambiando su contenido.

Tipos de datos

Al final hay que liberar la asignación con MPI_Type_free (newtype)

C

```
MPI_Type_free (&datatype)
```

Desasigna el objeto de tipo de datos especificado. El uso de esta rutina es especialmente importante para evitar el agotamiento de memoria si se crean muchos objetos de tipo de datos.

Tipos de datos

Tipo indexado

Generaliza al anterior, permitiendo bloques de longitud variable y con desplazamientos variables.

C

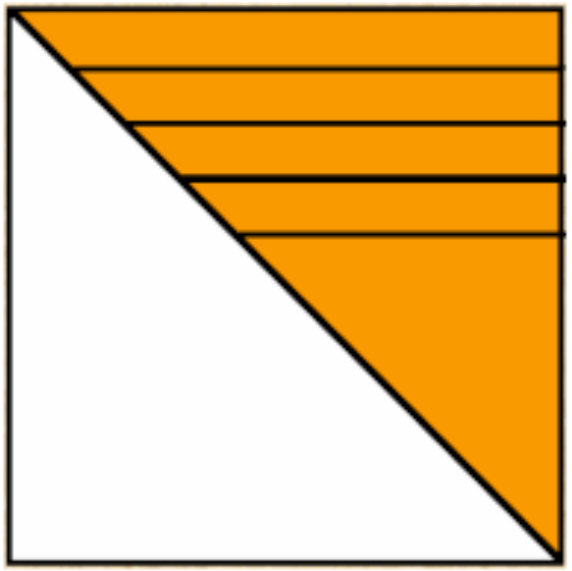
```
MPI_Type_indexed(cuantos, array-tambloque, array-desplazamientos,
                 oldtype, &newtype)
```

- cuantos: número de bloques.
- array-tambloque: array que contiene el **número de elementos** de cada bloque que forma el nuevo tipo.
- array-desplazamientos: es un array que contiene el desplazamiento necesario para acceder desde el inicio a cada bloque.
- oldtype: datatype antiguo
- newtype: nuevo tipo de dato

Tipos de datos

```
MPI_Type_indexed(cuantos, vec-largobloque, vec-desplazamiento,
                                        oldtype, &newtype)
```

Ejemplo.- Ejemplo: enviar el triángulo superior de una matriz de P0 a P1



```
float A[N][N], T[N][N];
MPI_Datatype T_tri;

for(i=0; i<N; i++) {
     long_bl[i] = N – i;
     desp[i] = (N+1) * i;
}

MPI_Type_indexed (N, long_bl, desp, MPI_FLOAT, &T_tri);
MPI_Type_commit (&T_tri);
     if (pid==0) MPI_Send(A, 1, T_tri, 1, 0, MPI_COMM_WORLD);
     else if (pid==1) MPI_Recv(T, 1, T_tri, 0, 0, MPI_COMM_WORLD, &info);
```

Tipos de datos

indexedtype.c

```
#include "mpi.h"
#include <stdio.h>
#define nelementos 6
int main(int argc, char *argv[])  {
int numtasks, rank, source=0, dest, tag=1, i;
int blocklengths[2], displacements[2];
float a[16] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
   9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[nelementos];
MPI_Status stat;
MPI_Datatype indextype;   // Nombre del nuevo tipo de dato

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

blocklengths[0] = 4;
blocklengths[1] = 2;
displacements[0] = 5;
displacements[1] = 12;
// Creamos el nuevo tipo de dato derivado indexed
MPI_Type_indexed(2,                // 2 bloques
                 blocklengths,     // array con el tamaño de cada bloque
                 displacements,    // array con los desplazamientos de cada bloque respecto del inicio
                 MPI_FLOAT,        // tipo de dato antiguo
                 &indextype);      // nuevo tipo de dato
MPI_Type_commit(&indextype);
if (rank == 0) {
  for (i=0; i<numtasks; i++)
   // Proceso 0 envia un elemento de tipo indextype a todos los procesos
     MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
  }

// todos los procesos reciben los datos indextype desde el proceso 0
MPI_Recv(b, nelementos, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
       rank,b[0],b[1],b[2],b[3],b[4],b[5]);
// Liberamos el tipo de dato cuando hemos acabado de utilizarlo
MPI_Type_free(&indextype);
MPI_Finalize();
}
```

displacements[0] = 5

displacements[1] = 12

1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0

blocklengths[0] = 4

blocklengths[1] = 2

```
mpicc -o indexedtype indexedtype.c
mpirun -np 4 ./indexedtype

rank= 0  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 1  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 2  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3  b= 6.0 7.0 8.0 9.0 13.0 14.0
```

Tipos de datos

`MPI_Type_create_struct`

Constructor de tipo de datos más general.

- Crea un nuevo tipo de datos que puede estar formado por bloques heterogéneos
  - Por ejemplo en C: estructuras

Tipos de datos

Como definir estructuras (objetos compuestos por varios tipos de datos) en C

C

```
struct mitipo
{
  int    ival;
  double dval[3];
};
```

Tipos de datos

`MPI_Type_create_struct`

C:

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

Tipos de datos

C:

```
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)
```

Parámetros:

- **count**: Un número entero que especifica el número de bloques. También es el número de entradas en los arrays array_of_types, array_of_displacements y array_of_blocklengths. (IN).
- **array_of_blocklengths:** El número de elementos en cada bloque (array de enteros). Es decir, array_of_blocklengths(i) especifica el número de elementos del tipo array_of_types(i) en el bloque(i). (IN).
- **array_of_displacements:** El desplazamiento en bytes de cada bloque desde el inicio de la estructura (array de enteros). (IN).
- **array_of_types:** El tipo de elementos en cada bloque. Es decir, el bloque(i) esta compuesto de una concatenación de tipo array_of_types(i). (IN).
- **newtype**: El nuevo tipo de datos (MPI_Datatype). (OUT).

Tipos de datos

C

```
struct compound
{
  int    ival;
  double dval[3];
};
```

---

count = 2
array_of_blocklengths[0] = 1
array_of_blocklengths[1] = 3
array_of_types[0] = MPI_INT
array_of_types[1] = MPI_DOUBLE

MPI_TYPE_CREATE_STRUCT (COUNT,
ARRAY_OF_BLOCKLENGTHS,
ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE)

Tipos de datos

Pero, ¿cómo calculamos los desplazamientos?

- Necesitamos crear una variable compuesta en nuestro programa.
- Calcular explícitamente las **direcciones de memoria de cada miembro**.
- **Restar direcciones** para obtener desplazamientos desde el origen.

Tipos de datos

¿Cómo determinar la dirección de un elemento?

`MPI_Get_address`

MPI_Get_address devuelve la dirección de una ubicación en memoria.

C:

```
int MPI_Get_Address (void *location,
                     MPI_Aint *address)
```

Tipos de datos

C

Ejemplo

```
struct PartStruct {
    char c;
    double d[6];
    int b[7];
} particle[100];
```

```
MPI_Datatype ParticleType;
int count = 3;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];

MPI_Get_address(&particle[0].c, &disp[0]);
MPI_Get_address(&particle[0].d, &disp[1]);
MPI_Get_address(&particle[0].b, &disp[2]);
/* Desplazamientos */
disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;

MPI_Type_create_struct (count, blocklen, disp, type, &ParticleType);
MPI_Type_commit (&ParticleType);

MPI_Send(particle,100,ParticleType,dest,tag,comm);
MPI_Type_free (&ParticleType);
```

Tipos de datos

```c
#include "mpi.h"
#include <stdio.h>
struct Partstruct
{
    char c;
    double d[6];
    int b[7];
};
int main(int argc, char *argv[])
{
    struct Partstruct particle[1000];
    int i, j, myrank;
    MPI_Status status;
    MPI_Datatype Particletype;
    MPI_Datatype type[3] = { MPI_CHAR, MPI_DOUBLE, MPI_INT };
    int blocklen[3] = { 1, 6, 7 };
    MPI_Aint disp[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
    {
        for (i=0; i<1000; i++) {
            particle[i].c = 'a';
            for (j=0; j<6; j++) {
                particle[i].d[j] = j;
            }
            for (j=0; j<7; j++) {
                particle[i].b[j] = j;
            }
        }
    }
    MPI_Get_address(&particle[0].c, &disp[0]);
    MPI_Get_address(&particle[0].d, &disp[1]);
    MPI_Get_address(&particle[0].b, &disp[2]);
    /* Desplazamientos */
    disp[2] -= disp[0]; disp[1] -= disp[0]; disp[0] = 0;
    MPI_Type_create_struct(3, blocklen, disp, type, &Particletype);
    MPI_Type_commit(&Particletype);
    if (myrank == 0)
    {
        MPI_Send(particle, 2, Particletype, 1, 123, MPI_COMM_WORLD);
    }
    else if (myrank == 1)
    {
        MPI_Recv(particle, 2, Particletype, 0, 123, MPI_COMM_WORLD, &status);
printf( "particle 0 c =  %c\n", particle[0].c );
printf("particle 0 d =  %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n", particle[0].d[0], particle[0].d[1],particle[0].d[2],particle[0].d[3],particle[0].d[4],particle[0].d[5]);
printf("particle 0 b =  %3d %3d %3d %3d %3d %3d %3d\n", particle[0].b[0], particle[0].b[1],particle[0].b[2],particle[0].b[3],particle[0].b[4],particle[0].b[5],particle[0].b[6]);
printf( "particle 1 c =  %c\n", particle[1].c );
printf("particle 1 d =  %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n", particle[1].d[0], particle[1].d[1],particle[1].d[2],particle[1].d[3],particle[1].d[4],particle[1].d[5]);
printf("particle 1 b =  %3d %3d %3d %3d %3d %3d %3d\n", particle[1].b[0], particle[1].b[1],particle[1].b[2],particle[1].b[3],particle[1].b[4],particle[1].b[5],particle[1].b[6]);
    }
    MPI_Type_free(&Particletype);
    MPI_Finalize();
    return 0;
}
```

struct_example.c

```
mpicc -o struct_example struct_example.c
mpirun -np 2 struct_example
particle 0 c =  a
particle 0 d =  0.0 1.0 2.0 3.0 4.0 5.0
particle 0 b =    0   1   2   3   4   5   6
particle 1 c =  a
particle 1 d =  0.0 1.0 2.0 3.0 4.0 5.0
particle 1 b =    0   1   2   3   4   5   6
```