MESSAGE PASSING INTERFACE MPI (PARTE B)

Adaptado de transparencias de Thomas Sterling y Chirag Dekate

Temas

- Llamadas colectivas MPI
  - Primitivas de Sincronización
  - Primitivas de Comunicación
  - Primitivas de Reducción
- Tipos de datos derivados:
  - Introducción
  - Contiguous
  - Vector
  - Indexed
  - Struct
- Ejemplo: Multiplicación de una Matriz por un Vector

Repaso

- 6 funciones principales:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- MPI Header
  - #include “mpi.h”
- Tipos de datos básics
  - MPI_INT, MPI_FLOAT, ….

Llamadas colectivas (Collective Calls)

- Una **comunicación colectiva** abarca todos los procesos de un comunicador.
- MPI posee varias funciones de comunicación colectiva:
  - Sincronización
    - Barrier
  - Comunicación
    - Broadcast
    - Gather & Scatter
    - All Gather
  - Reducción
    - Reduce
    - AllReduce

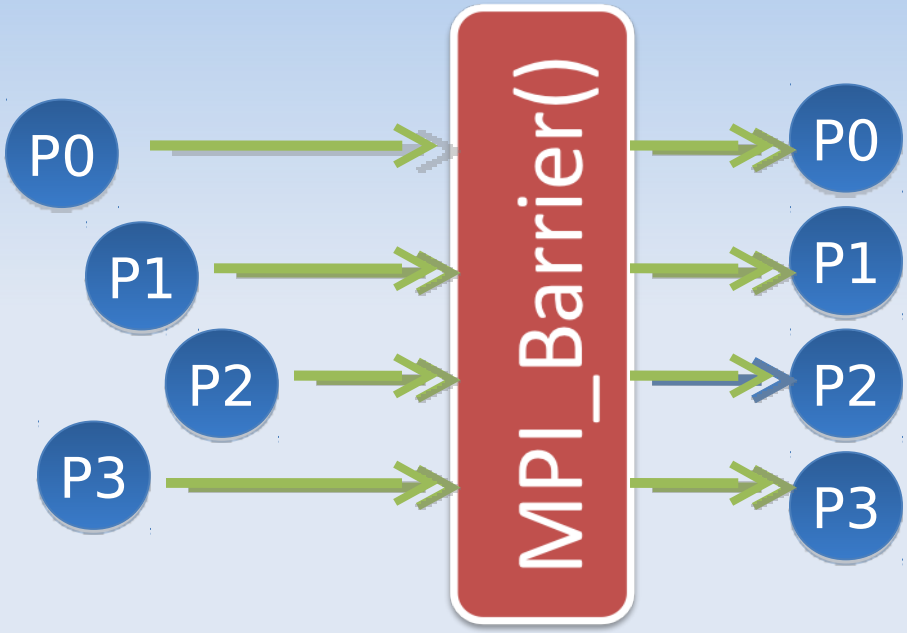
Barreras:

Función: MPI_Barrier()

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

Descripción:

Crea una barrera en el grupo del comunicacor *comm*. Cada proceso, al llegar a la barrera, bloquea hasta que todos los proceso alcancen la misma barrera.



http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Barrier.html

Ejemplo: MPI_Barrier()

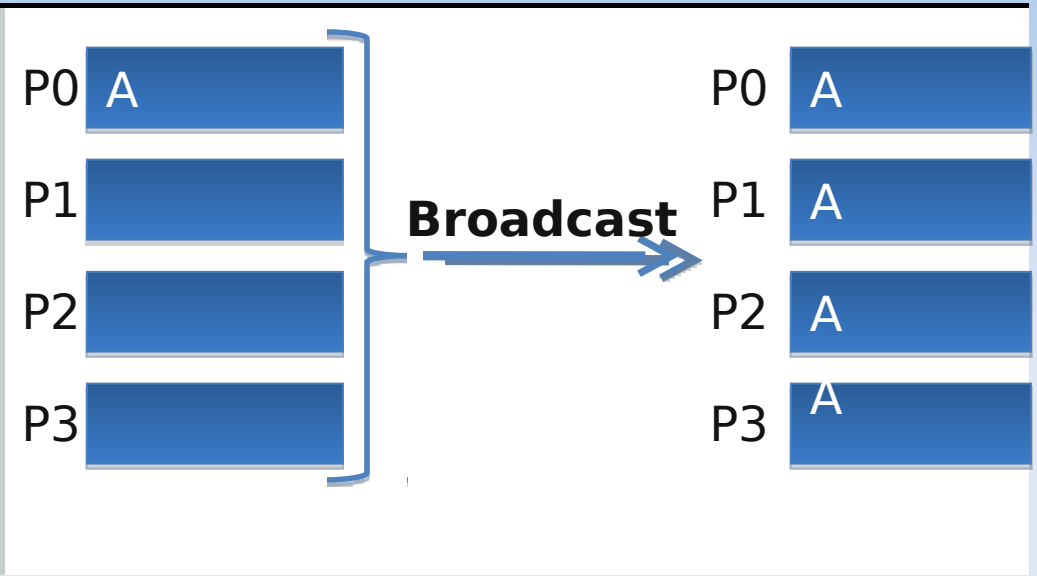
```
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv){
  int rank, num_processes, sleep_time;
  MPI_Init(&argc, &argv);
  sleep_time = argc > 1 ? atoi(argv[1]) : 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
  printf("Proceso %d, antes de la barrera\n", rank);
  if (rank == 0) sleep(sleep_time);
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Proceso %d, despues de la barrera\n", rank);

  return MPI_Finalize();
}
```

```
$ $ mpirun -np 4 ./barrier
Proceso 1, antes de la barrera
Proceso 2, antes de la barrera
Proceso 3, antes de la barrera
Proceso 0, antes de la barrera
Proceso 0, despues de la barrera
Proceso 2, despues de la barrera
Proceso 1, despues de la barrera
Proceso 3, despues de la barrera
```

Broadcast

Función: MPI_Bcast()

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



Descripción:

Es una llamada colectiva en la que un proceso envía los datos contenidos en *message* a todos los procesos en el comunicador. Todos los procesos invocan a *MPI_Bcast* con los mismos argumentos para root y *comm*,

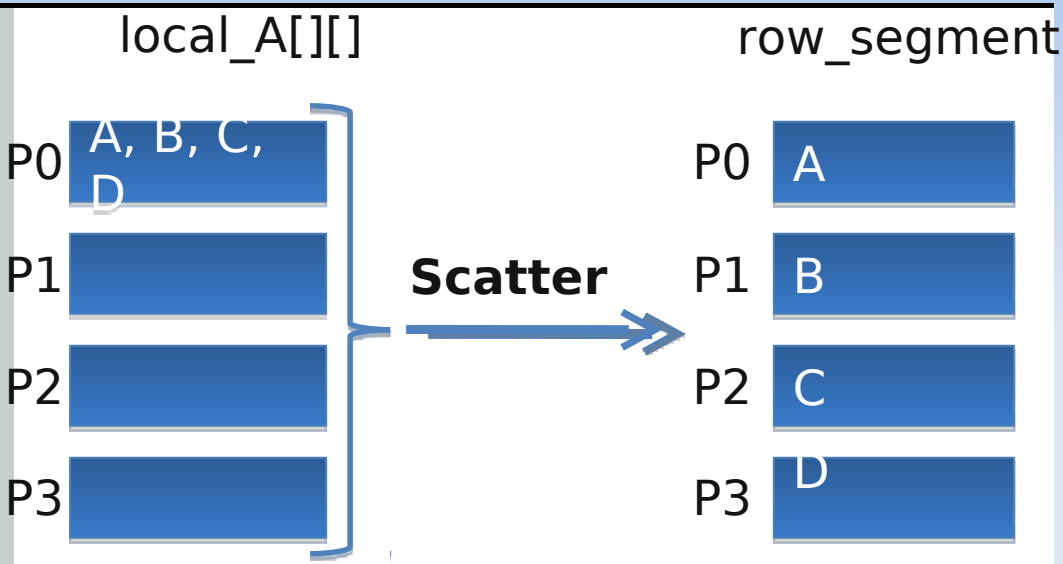
```
float       endpoint[2];
...
MPI_Bcast(endpoint, 2, MPI_FLOAT, 0, MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Bcast.html

Scatter

Función: MPI_Scatter()

```
int MPI_Scatter (
        void            *sendbuf,
        int             send_count,
        MPI_Datatype    send_type,
        void            *recvbuf,
        int             recv_count,
        MPI_Datatype    recv_type,
        int             root,
        MPI_Comm        comm)
```



Descripción:

MPI_Scatter divide los datos almacenados en *sendbuf* por el proceso con rango *root* en *p* segmentos compuestos por *send_count* elementos de tipo *send_type*. El primer segemento se envía al proceso 0, el segundo al proceso 1, etc. Los argumentos *send* son significativos sólo en el proceso con rango *root*.

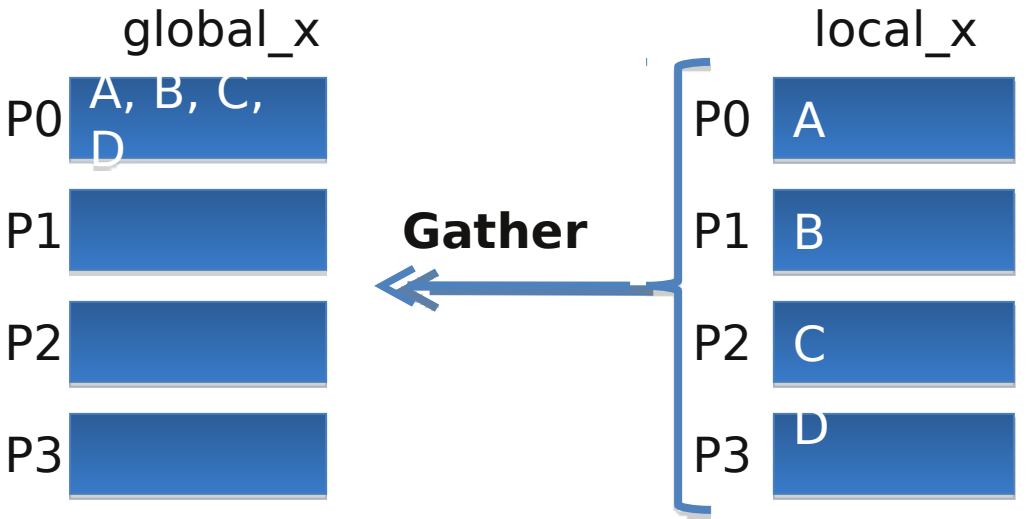
```
...
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT, row_segment, n/p, MPI_FLOAT, 0,MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Scatter.html

Gather

Función: MPI_Gather()

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



Descripción:

MPI_Gather reúne los datos referenciados por *sendbuf* en cada proceso del comunicador *comm,* y almacena los datos en orden de rango en la posición *recvbuf* del proceso con rango *root.* Los parámetros *recv* son sólo significativos en dicho proceso..

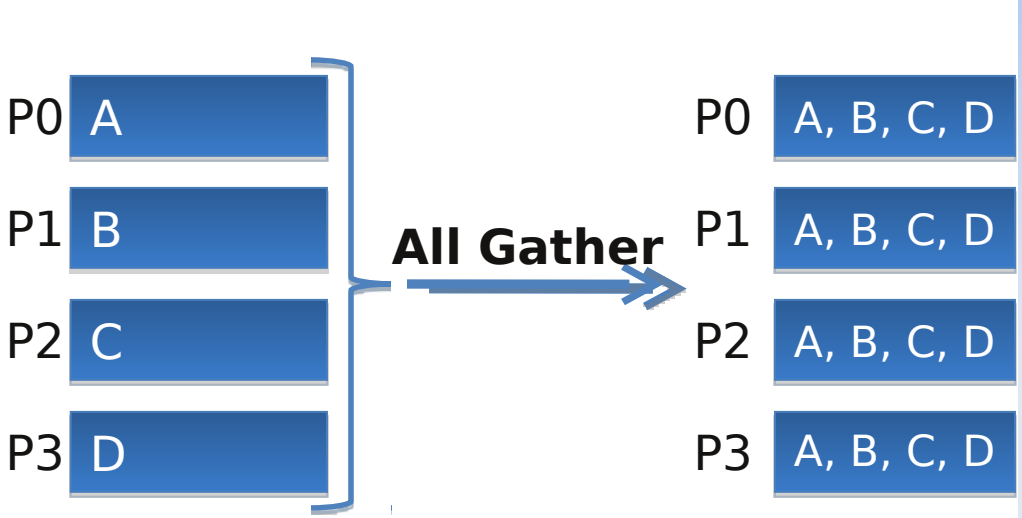
```
...
    MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p, MPI_FLOAT, 0, MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Gather.html

All Gather

Función: MPI_Allgather()

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



Descripción:

MPI_Allgather recoge el contenido del buffer *sendbuf* en cada proceso. El efecto es similar a ejecutar MPI_Gather() p veces con diferentes procesos actuando como *root*.

```
 for (root=0; root<p; root++)
    MPI_Gather(local_x, n/p, MPI_FLOAT, global_x, n/p, MPI_FLOAT, root, MPI_COMM_WORLD);
...
```

Puede reemplazarse con :

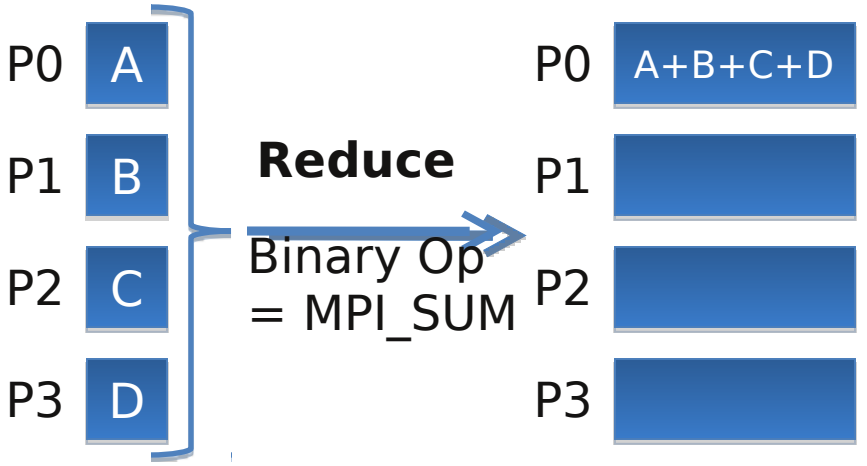
```
MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT, MPI_COMM_WORLD);
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Allgather.html

Reduce

Función: MPI_Reduce()

```
int MPI_Reduce (
    void          *operand,
    void          *result,
    int           count,
    MPI_Datatype  datatype,
    MPI_Op        operator,
    int           root,
    MPI_Comm      comm )
```



Descripción:

Es una llamada colectiva en la que todos los procesos en un comunicador contribuyen datos que se combinan usando una operación binaria (MPI_Op) tal como suma, máximo, mínimo, etc. MPI_Reduce combina los operandos almacenados en la memoria referenciada por *operand* usando la operación *operator* y almacena el resultado en **result*. Todos los procesos en el comunicador *comm* invocan a MPI_Reduce con los mismos valores en *count, datatype operator* y *root.*

```
...
MPI_Reduce(&local_integral, &integral, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Reduce.html

Operadores binarios

- Se usan en las llamadas a MPI_Reduce como uno de los parámetros. MPI_Reduce realiza una reducción global, determinada por el dicho operador, en los operandos provistos.
- Algunos de los operdores binarios son :

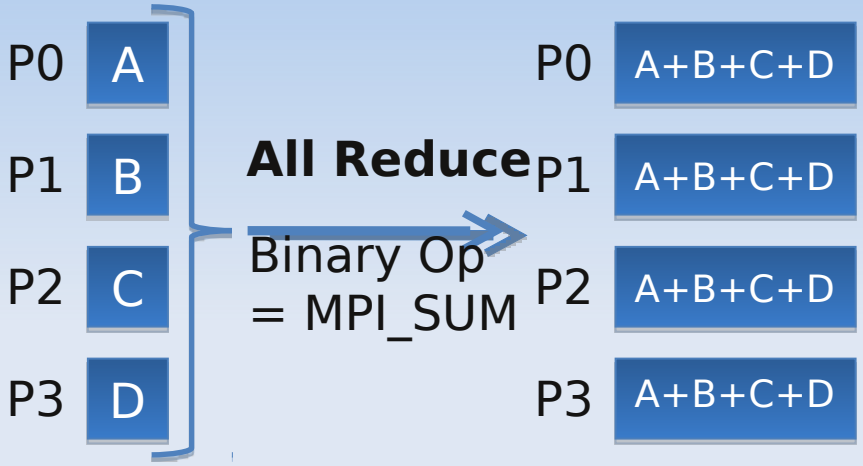
Operation Name	Meaning
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	Y Lógico
MPI_BAND	Y “bitwise”
MPI_LOR	O Lógico
MPI_BOR	O “bitwise”
MPI_LXOR	XOR Lógico
MPI_BXOR	XOR “bitwise”
MPI_MAXLOC	Máximo y ubicación del máximo.
MPI_MINLOC	Mínimo y ubicación del mínimo.

```
MPI_Reduce(&local_integral, &integral, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
```

All Reduce

Función: MPI_Allreduce()

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



Descripción:

MPI_Allreduce se usa como MPI_Reduce, pero el resultado de la reducción se devuelve en todos los procesos, por lo que no existe el parámetro *root*.

```
...
MPI_Allreduce(&integral, &integral, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Allreduce.html

Nuevos tipos de datos

- Creación de estructuras de datos en C:

```
typedef struct {
        . . .
} STRUCT_NAME
```

- Por ejemplo, podríamos crear la siguiente estructura para utilizar en el programa que implementa la regla del trapecio:

```
typedef struct {
    float a,
    float b,
    int n;
} DATA_INTEGRAL;
. . .
. . .
DATA_INTEGRAL intg _data;
```

- Pero NO podemos hacer:

```
MPI_Bcast( &intg_data, 1, DATA_INTEGRAL, 0,
MPI_COMM_WORLD);
```

¡ERROR!
DATA_INTEGRAL
NO es un
MPI_Datatype

Construcción de MPI Datatypes

- MPI permite definir tipos derivados, a partir de los tipos básicos.
- Estos tipos derivados pueden usarse en las funciones de comunicación, en lugar de los tipos básicos.
- Un proceso emisor puede empaquetar datos no contiguos en un buffer contiguo y enviar el contenido del buffer a un proceso receptor que desempaquetará el buffer contiguo y lo almacenará en forma no contigua.
- Un tipo derivado es un objeto opaco que especifica:
  - Una secuencia de tipos primitivos
  - Una secuencia de desplazamientos enteros (bytes)
- Hay varias formas de construir tipos de datos derivados:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Tipos de datos básicos

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Tipos derivados : Contiguous

Función: MPI_Type_contiguous()

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

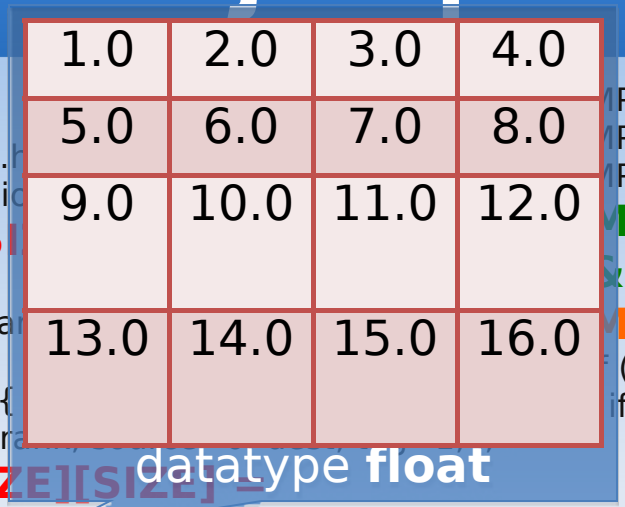
Descripción:

Es el constructor más simple. Crea un nuevo tipo de datos consistente en *count* copias de un tipo existente (*old_type*)

```
MPI_Datatype rowtype;
...
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Type_contiguous.html

Ejemplo : Contiguous




```
#include "mpi.h
#include <stdio
#define SI

int main(argc,a
int argc;
char *argv[];  {
int numtasks, r
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;
```

```
PI_Init(&argc,&argv);
PI_Comm_rank(MPI_COMM_WORLD, &rank);
PI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT,
&rowtype);
MPI_Type_com
(numtasks == SIZ
if (rank == 0) {
  for (i=0; i<numta
      dest = i;
      MPI_Send
tag, MPI_COM
    }
  }
  MPI_Recv(b,
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("Must specify %d processors. Terminating.\n",SIZE);
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Vector

Función: MPI_Type_vector()

```
int MPI_Type_vector(
        int count,
        int blocklen,
        int stride,
        MPI_Datatype old_type,
        MPI_Datatype *newtype )
```

Descripción:

Devuelve un nuevo tipo de datos que representa bloques igualmente espaciados. El espaciado entre los comienzos de cada bloque se expresa en unidades del tamaño de *old_type*. El parámetro *count* representa el número de bloques, *blocklen* es el número de elementos en cada bloque, y *stride* representa el número de elementos entre los comienzos de bloque. El nuevo tipo se almacena en *new_type*

```
...
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Type_vector.html

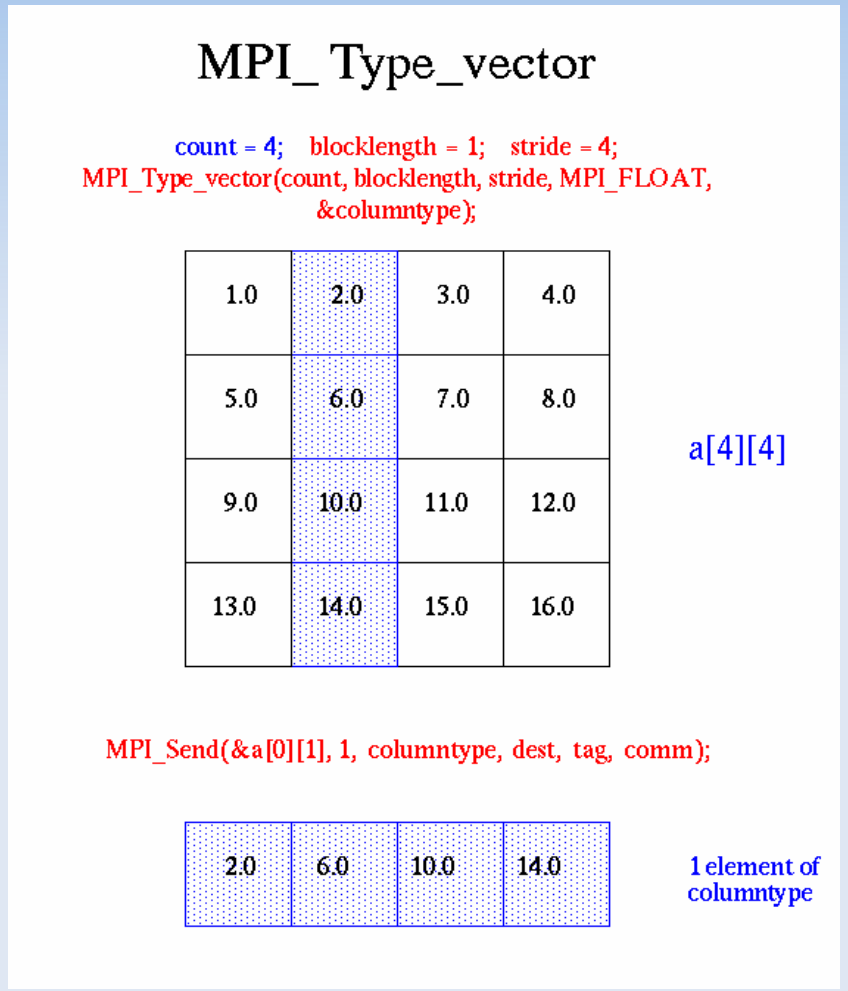
Ejemplo: Vector

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

int main(argc,argv)
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;
```

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

MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT,
&columntype);
MPI_Type_commit(&columntype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
        MPI_Send(&a[0][i], 1, columntype, i, tag,
MPI_COMM_WORLD);
      }

   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("Must specify %d processors. Terminating.\n",SIZE);

MPI_Type_free(&columntype);
MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Ejemplo : Vector



https://computing.llnl.gov/tutorials/mpi/

Indexado

Función: MPI_Type_indexed()

```
int MPI_Type_indexed(
        int count,
        int *array_of_blocklengths,
        int *array_of_displacements,
        MPI_Datatype oldtype,
        MPI_datatype *newtype);
```

Descripción:

Devuelve un nuevo tipo que representa *count* bloques. Cada bloque está definido por una entrada en *array_of_blocklengths* y *array_of_displacements*. Los desplazamientos se expresan en unidades de tamaño *oldtype*. El parámetro *count* es el número de bloques, igual al número de entradas de *array_of_displacements* (desplazamiento de cada bloque en unidades de tipo *oldtype*) y de *array_of_blocklengths* (número de instancias de *oldtype* en cada bloque).

```
...
MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
...
```

https://computing.llnl.gov/tutorials/mpi/man/MPI_Type_indexed.txt

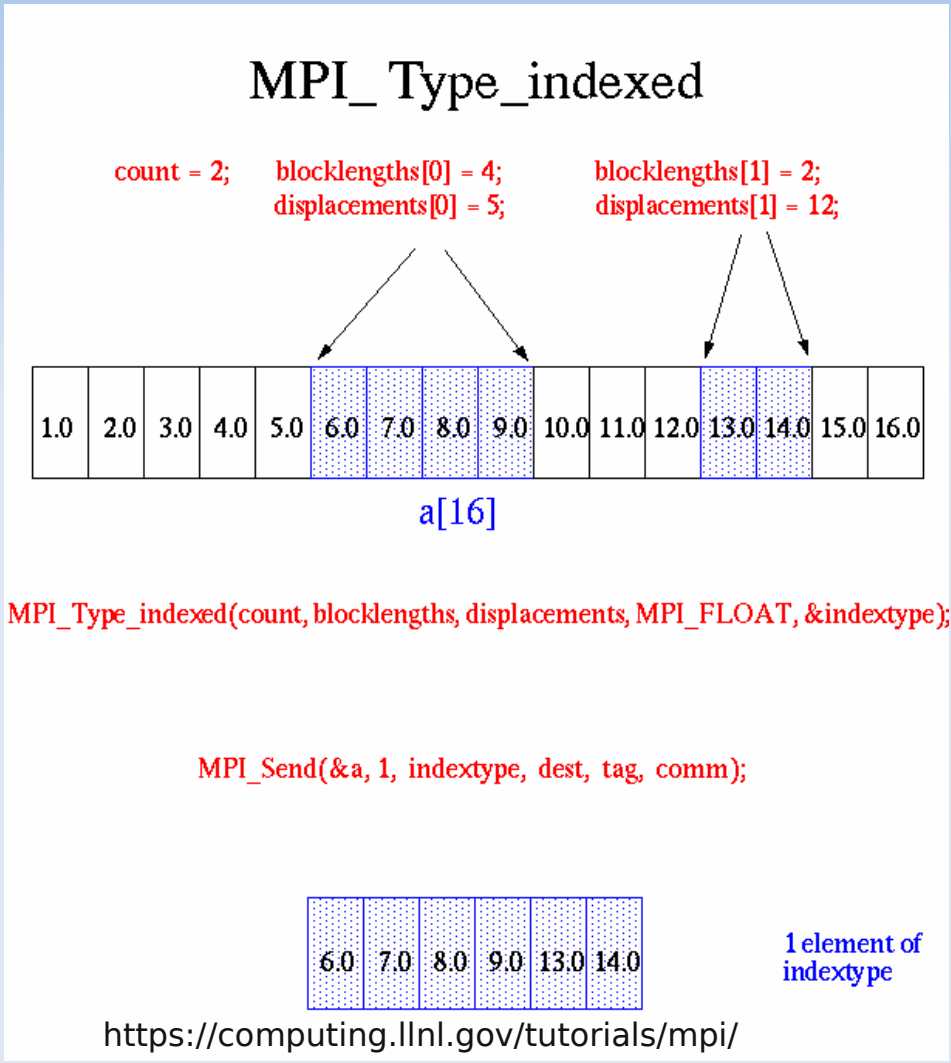
Ejemplo : Indexado

```
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

int main(argc,argv)
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[NELEMENTS];

MPI_Status stat;
MPI_Datatype indextype;

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;

MPI_Type_indexed(2,
blocklengths, displacements,
MPI_FLOAT, &indextype);
MPI_Type_commit(&indextype);

if (rank == 0) {
  for (i=0; i<numtasks; i++)
      MPI_Send(a, 1, indextype, i,
tag, MPI_COMM_WORLD);
  }

MPI_Recv(b, NELEMENTS,
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]);

MPI_Type_free(&indextype);
MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Ejemplo: Indexado



Estructuras

Función: MPI_Type_struct()

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

Descripción:

Devuelve un nuevo tipo que representa *count* bloques. Cada bloque está definido por una entrada en *array_of_blocklengths*, *array_of_displacements* y *array_of_types*. Los deplazamientos se expresan en bytes. *count* es un entero que especifica el número de bloques (número de entradas en los arreglos). *array_of_blocklengths* es el número de elementos en cada blouqe, y *array_of_displacements* especifica el desplazamiento en bytes de cada bloque. *array_of_types* indica el tipo de datos que compone cada bloque.

```
...
MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
...
```

https://computing.llnl.gov/tutorials/mpi/man/MPI_Type_struct.txt

Ejemplo: Estructuras

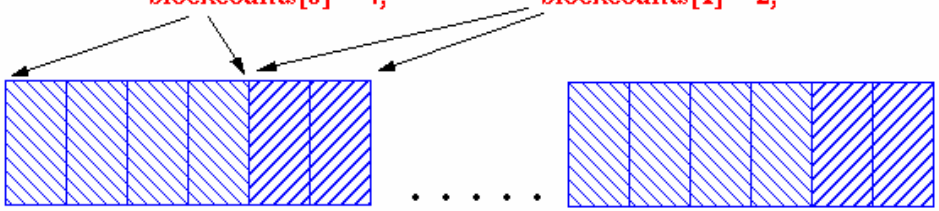
```
#include "mpi.h"
#include <stdio.h>
#define NELEM 25
int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, source=0, dest, tag=1, i;
typedef struct {
  float x, y, z;
  float velocity;
  int  n, type;
  }        Particle;
Particle     p[NELEM], particles[NELEM];
MPI_Datatype particletype, oldtypes[2];
int          blockcounts[2];

/* MPI_Aint type used to be consistent with syntax of */
/* MPI_Type_extent routine */
MPI_Aint    offsets[2], extent;

MPI_Status stat;

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

/* Setup description of the 4 MPI_FLOAT
fields x, y, z, velocity */
offsets[0] = 0;
oldtypes[0] = MPI_FLOAT;
blockcounts[0] = 4;

MPI_Type_extent(MPI_FLOAT, &extent);
offsets[1] = 4 * extent;
oldtypes[1] = MPI_INT;
blockcounts[1] = 2;
MPI_Type_struct(2, blockcounts, offsets,
oldtypes, &particletype);
MPI_Type_commit(&particletype);

if (rank == 0) {
  for (i=0; i<NELEM; i++) {
     particles[i].x = i * 1.0;
     particles[i].y = i * -1.0;
     particles[i].z = i * 1.0;
     particles[i].velocity = 0.25;
     particles[i].n = i;
     particles[i].type = i % 2;
    }
  for (i=0; i<numtasks; i++)
    MPI_Send(particles, NELEM, particletype, i, tag,
MPI_COMM_WORLD);
  }
MPI_Recv(p, NELEM, particletype, source, tag,
MPI_COMM_WORLD, &stat);
printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n",
rank,p[3].x,
    p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

MPI_Type_free(&particletype);
MPI_Finalize();
}
```

https://computing.llnl.gov/tutorials/mpi/

Ejemplo: Estructuras

MPI_ Type_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;   oldtypes[1] = MPI_INT
           offsets[0] = 0;            offsets[1] = 4 * extent;
           blockcounts[0] = 4;        blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

https://computing.llnl.gov/tutorials/mpi/

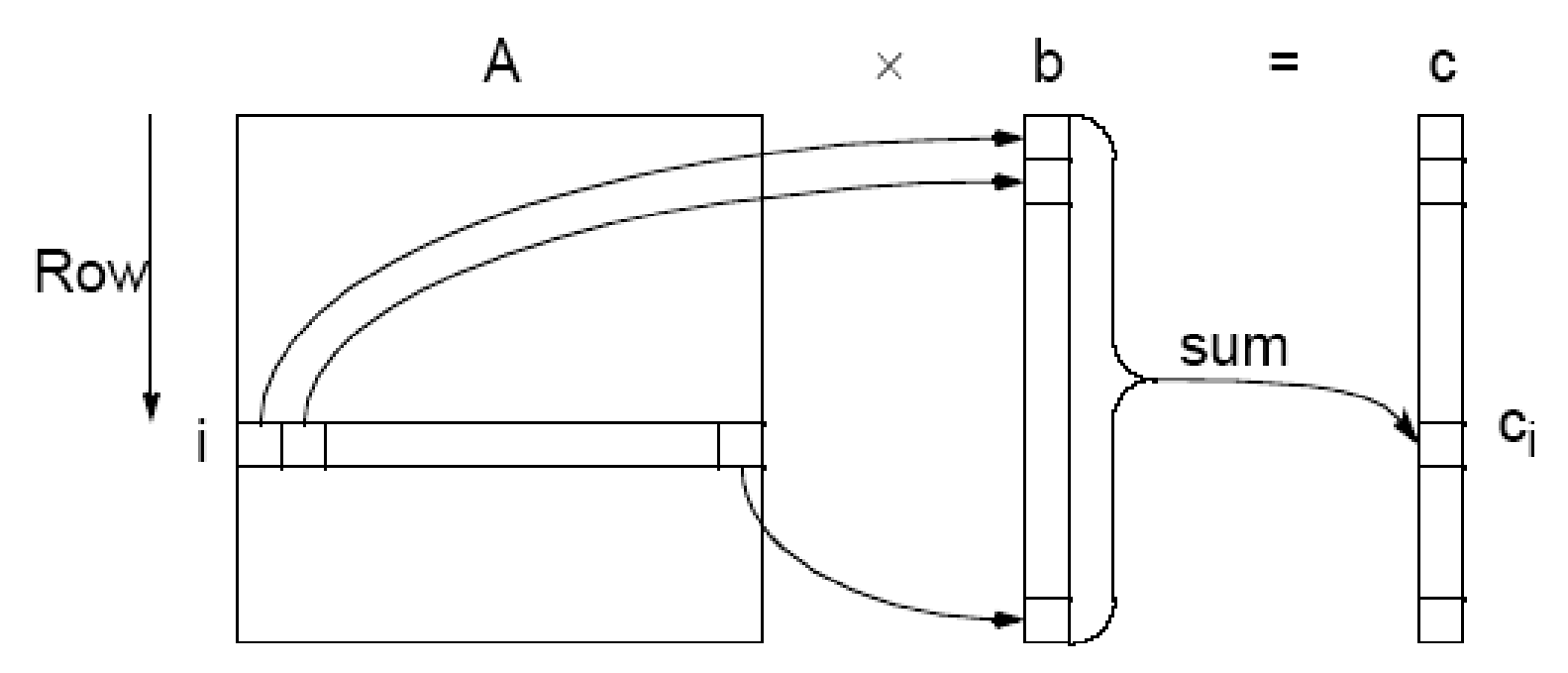
Multiplicación de una matriz por un vector

La multiplicación de una matriz, **A** pot un vector **B**, produce el vector **C**, dode los c_i ($0 <= i < n$), se computan de la siguiente forma:

$$C_i = \sum_{k=0}^{k=m} A_{ik} * B_k$$

A es una matriz *n x m*, **B** es un vector de tamaño *m* y **C** es un vector de tamaño *n*.

Multiplicación de una matriz por un vector



Profiling : MPI_Wtime

Función: MPI_Wtime()

double MPI_Wtime()

Descripción:

Returns time in seconds elapsed on the calling processor. Resolution of time scale is determined by the MPI environment variable MPI_WTICK. When the MPI environment variable MPI_WTIME_IS_GLOBAL is defined and set to true, the the value of MPI_Wtime is synchronized across all processes in MPI_COMM_WORLD

```
 double time0;
...
 time0 = MPI_Wtime();
...
 printf("Hello From Worker #%d %lf \n", rank, (MPI_Wtime() - time0));
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Wtime.html