MPI : Message Passing Interface

M.I. Raymundo Antonio González Grimaldo

Fecha

MPI

- MPI ("Message Passing Interface", Interfaz de Paso de Mensajes)
- Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca
- Diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

MPI

- El esfuerzo para estandarizar MPI involucró a cerca de 60 personas de 40 organizaciones diferentes principalmente de EE.UU. y Europa.
- Con MPI el número de procesos requeridos se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta.

MPI

- A cada proceso se le asigna una variable que se denomina **rank**, la cual identifica a cada proceso, en el rango de 0 a p-1, donde **p** es el número total de procesos.
- El control de la ejecución del programa se realiza mediante la variable rank; la variable rank permite determinar que proceso ejecuta determinada porción de código.
- En MPI se define un comunicador como una colección de procesos, los cuales pueden enviar mensajes el uno al otro; el comunicador básico se denomina MPI_COMM_WORLD

Funciones

Header	
C include file	**Fortran include file**
`#include "mpi.h"`	`include 'mpif.h'`

Formato de Funciones	
Formato	`rc = MPI_Xxxxx(parameter, ... )`
Ejemplo	`rc = MPI_Bsend(&buf,count,type,dest,tag,comm)`
Código de Error	`Regresa en "rc". MPI_SUCCESS si fue exitoso`

Funciones básicas

MPI_Init permite inicializar una sesión MPI.

- Esta función debe ser utilizada antes de llamar a cualquier otra función de MPI.

MPI_Finalize permite terminar una sesión MPI.

- Esta función debe ser la última llamada a MPI que un programa realice.
- Permite liberar la memoria usada por MPI.

MPI_Comm_size permite determinar el número total de procesos que pertenecen a un comunicador.

MPI_Comm_rank permite determinar el identificador (rank) del proceso actual.

Programa1

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

int main (int argc, char *argv[])
{
    int name,p;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &name);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    printf("Hola mundo, desde el proceso %d de %d \n",name,p);
    MPI_Finalize();
  return 0;
}
```

Hola Mundo!!!

Point-to-Point Communication

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

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

Point-to-Point Communication

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of receive buffer
IN	count	max number of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status

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

Send Buffer and Message Data

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

Programa 2

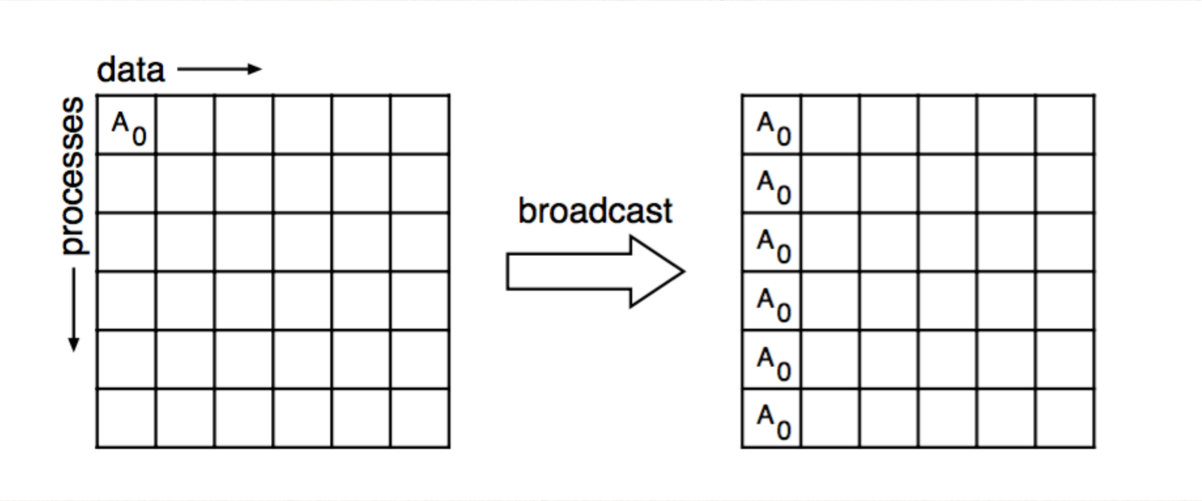
Suma de los elementos de un arreglo

MPI_Sendrecv

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	sendbuf	initial address of send buffer
IN	sendcount	number of entries to send
IN	sendtype	type of entries in send buffer
IN	dest	rank of destination
IN	sendtag	send tag
OUT	recvbuf	initial address of receive buffer
IN	recvcount	max number of entries to receive
IN	recvtype	type of entries in receive buffer
IN	source	rank of source
IN	recvtag	receive tag
IN	comm	communicator
OUT	status	return status

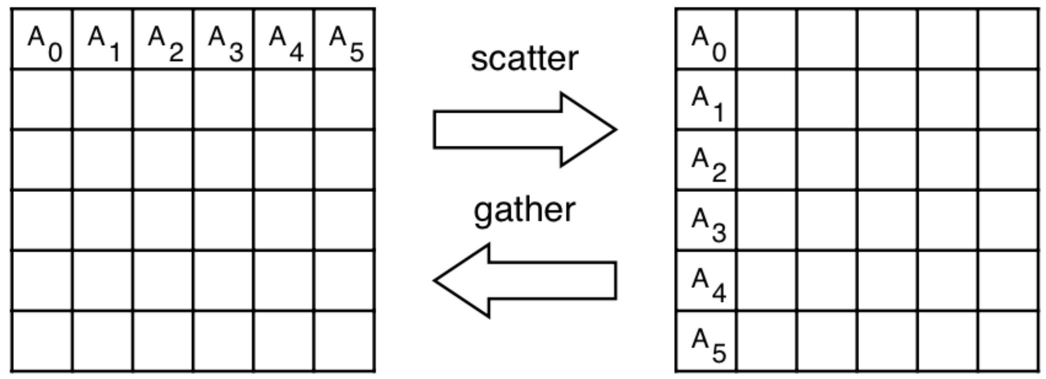
```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
              int dest, int sendtag, void *recvbuf, int recvcount,
              MPI_Datatype recvtype, int source,
              MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)
```



MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	starting address of buffer
IN	count	number of entries in buffer
IN	datatype	data type of buffer
IN	root	rank of broadcast root
IN	comm	communicator

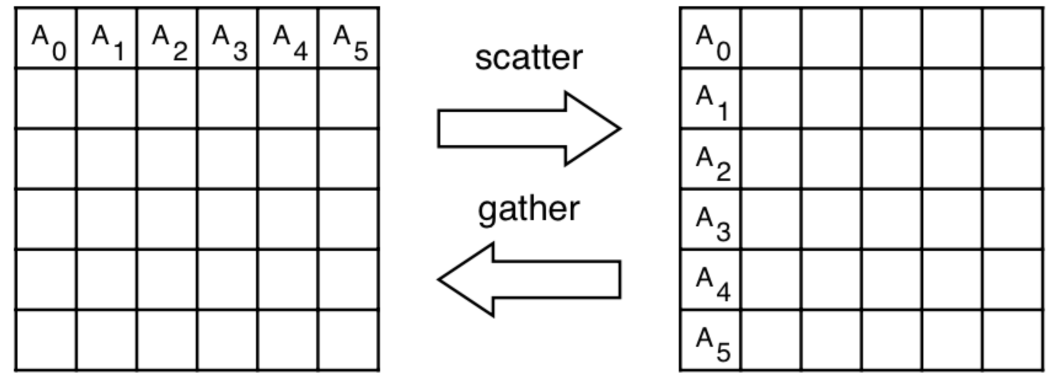
```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm )
```



MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

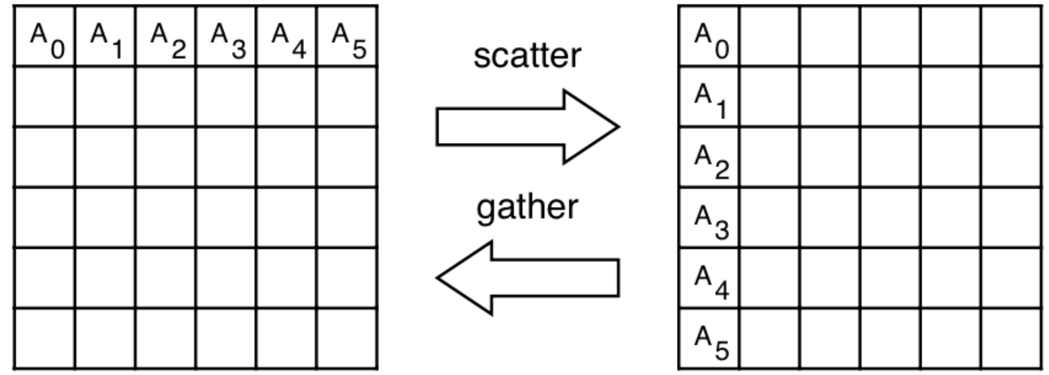
IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements for any single receive
IN	recvtype	data type of recv buffer elements
IN	root	rank of receiving process
IN	comm	communicator

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



Example 4.3 Previous example modified – only the root allocates memory for the receive buffer.

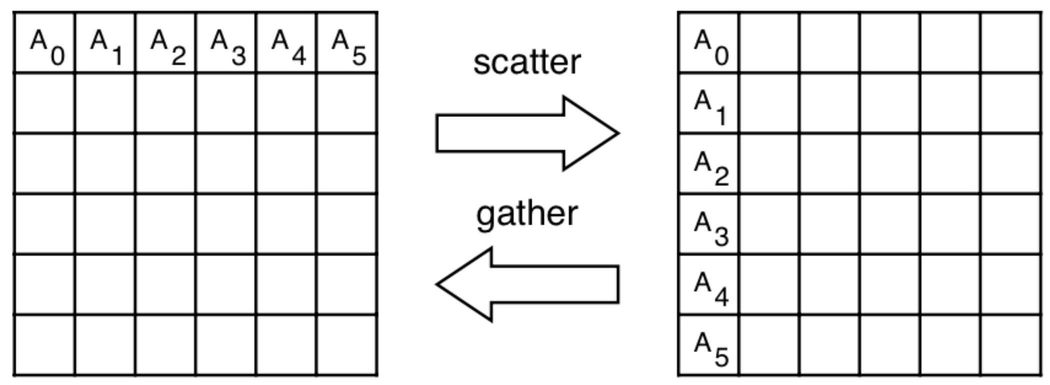
```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
   MPI_Comm_size( comm, &gsize);
   rbuf = (int *)malloc(gsize*100*sizeof(int));
   }
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

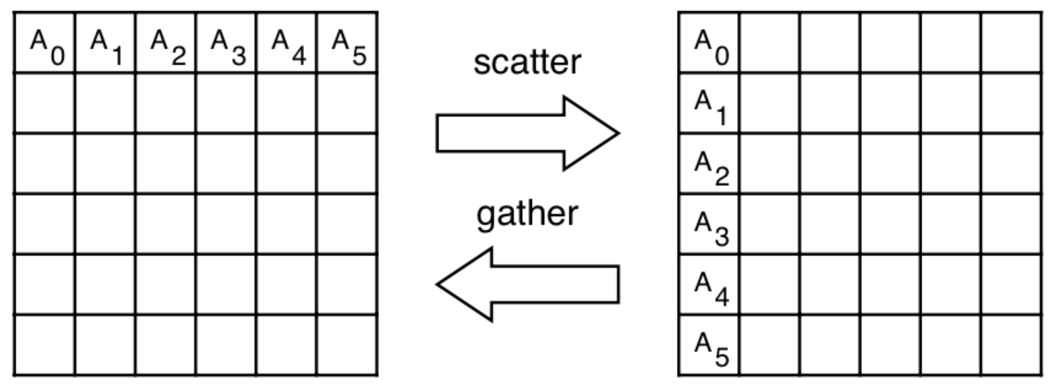
IN	sendbuf	address of send buffer
IN	sendcount	number of elements sent to each process
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process
IN	comm	communicator

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



Example 4.11 The reverse of Example 4.2, page 155. Scatter sets of 100 ints from the root to each process in the group. See Figure 4.7.

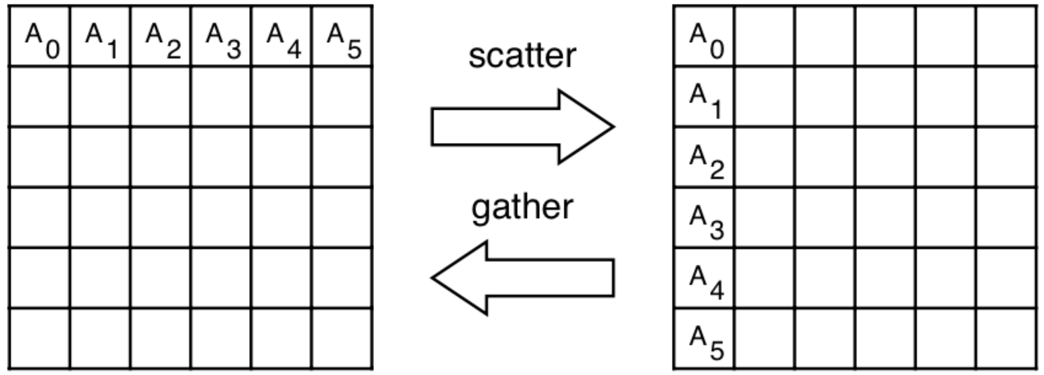
```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
...
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
...
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```



MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	sendbuf	starting address of send buffer
IN	sendcount	number of elements in send buffer
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcounts	integer array
IN	displs	integer array of displacements
IN	recvtype	data type of recv buffer elements
IN	root	rank of receiving process
IN	comm	communicator

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



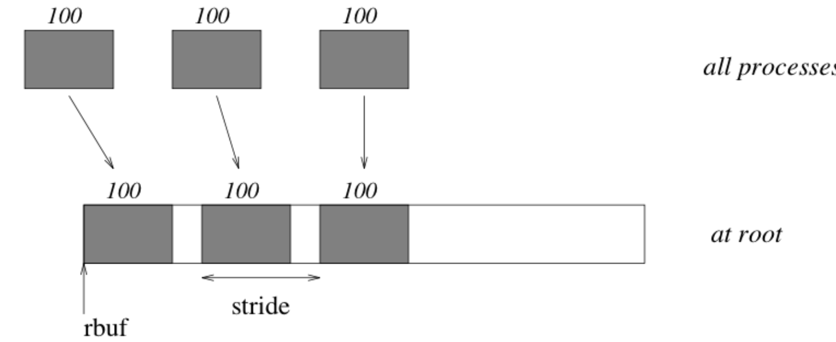


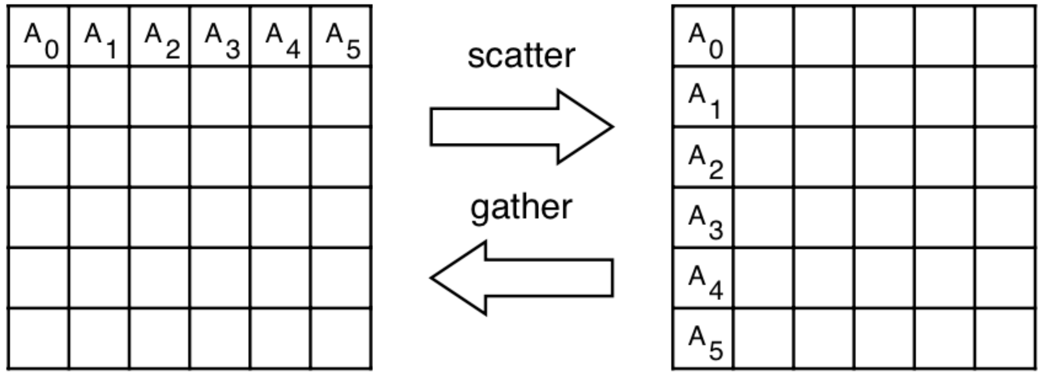
Figure 4.3
The root process gathers 100 **ints** from each process in the group, each set is placed **stride** ints apart.

Example 4.5 Have each process send 100 ints to root, but place each set (of 100) *stride* ints apart at receiving end. Use MPI_GATHERV and the displs argument to achieve this effect. Assume $stride \geq 100$. See Figure 4.3.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf, stride;
int *displs,i,*rcounts;

...

MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs, MPI_INT,
                                                         root, comm);
```



MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer
IN	sendcounts	integer array
IN	displs	integer array of displacements
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process
IN	comm	communicator

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Reduce

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of send buffer
OUT	recvbuf	address of receive buffer
IN	count	number of elements in send buffer
IN	datatype	data type of elements of send buffer
IN	op	reduce operation
IN	root	rank of root process
IN	comm	communicator

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

Reduce

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Definición de tipos de datos

Type Contiguous

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count
IN	oldtype	old datatype
OUT	newtype	new datatype

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

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

Type Contiguous

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

IN	count	replication count
IN	oldtype	old datatype
OUT	newtype	new datatype

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

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

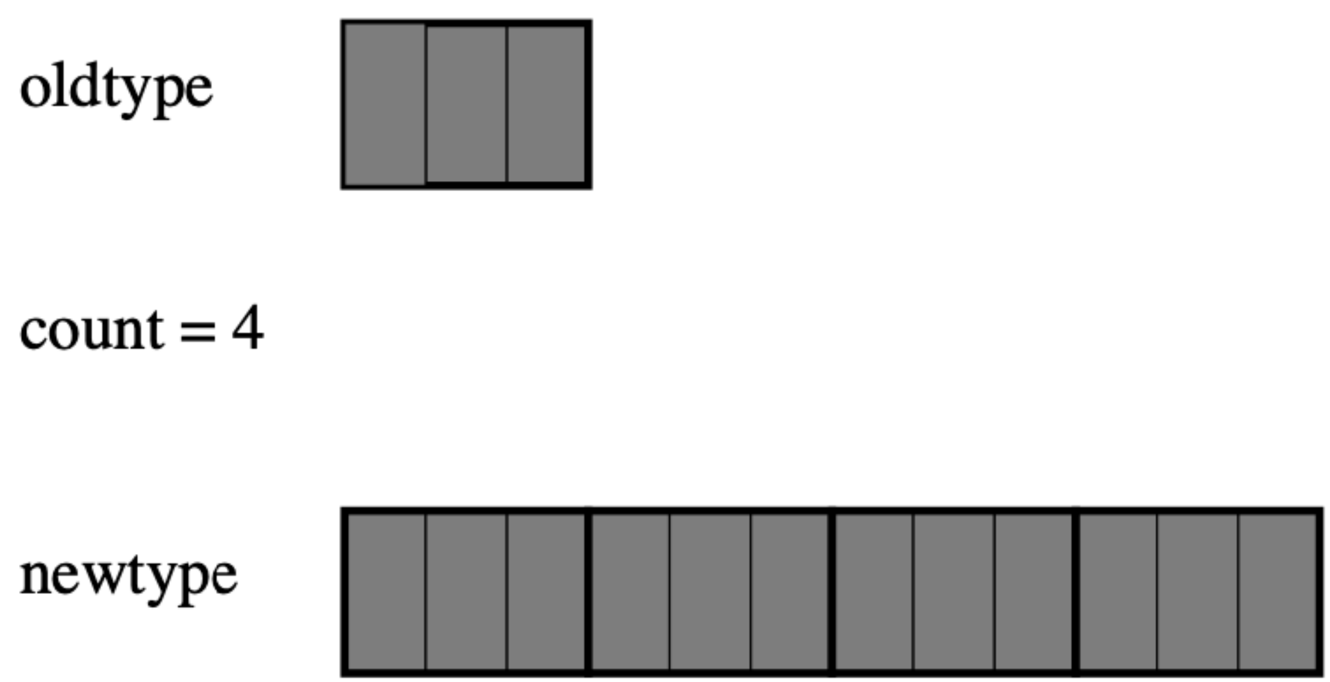


Figure 3.2
Effect of datatype constructor MPI_TYPE_CONTIGUOUS.

Type Commit

MPI_TYPE_COMMIT(datatype)

INOUT	datatype	datatype that is to be committed

```
int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Type Free

MPI_TYPE_FREE(datatype)

INOUT	datatype	datatype to be freed

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

```
MPI_TYPE_FREE(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Type Struct

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	byte displacement for each block
IN	array_of_types	type of elements in each block
OUT	newtype	new datatype

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

Type Struct

MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN	count	number of blocks
IN	array_of_blocklengths	number of elements per block
IN	array_of_displacements	byte displacement for each block
IN	array_of_types	type of elements in each block
OUT	newtype	new datatype

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

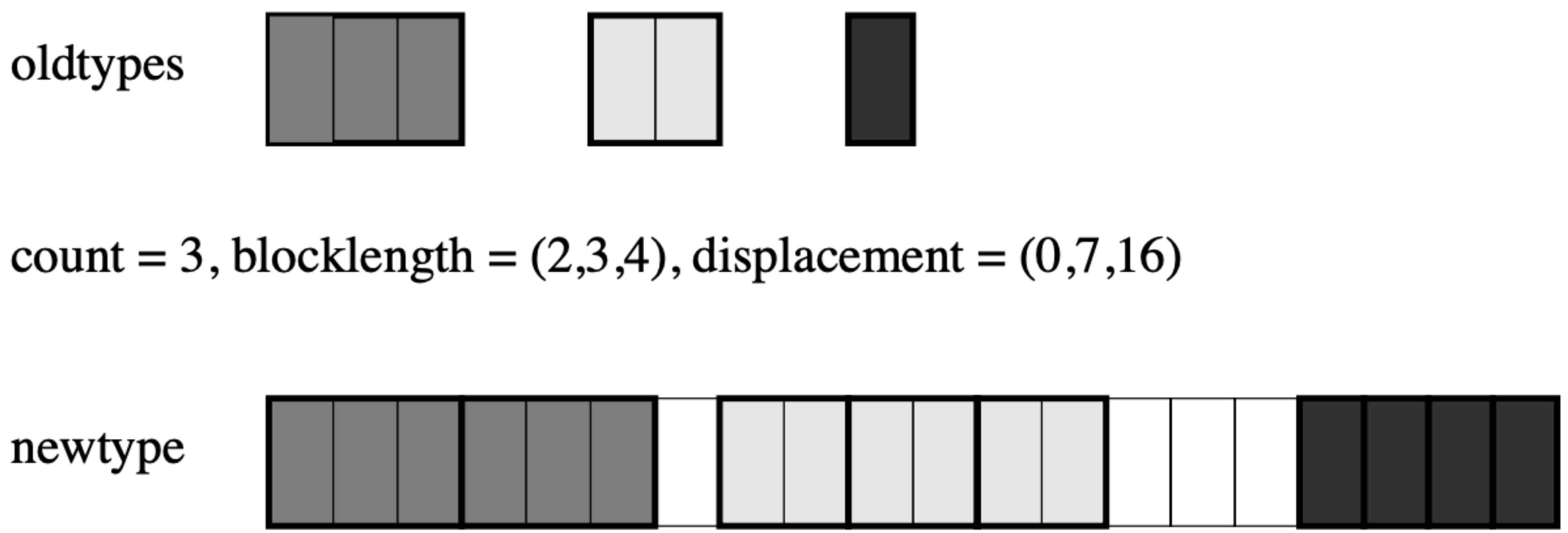


Figure 3.8
Datatype constructor MPI_TYPE_STRUCT.