**5.1. Производные типы данных**

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они не могут использоваться ни в каких других операциях, кроме коммуникационных. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. Производный тип MPI представляет собой скрытый (opaque) объект, который специфицирует две вещи: последовательность базовых типов и последовательность смещений. Последовательность таких пар определяется как *отображение (карта) типа*:

Typemap = {(type0, disp0), ... , (typen-1, dispn-1)}

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию. Отображение типа вместе с базовым адресом начала расположения данных buf определяет коммуникационный буфер обмена. Этот буфер будет содержать n элементов, а i-й элемент будет иметь адрес buf+disp и иметь базовый тип type. Стандартные типы MPI имеют предопределенные отображения типов. Например, MPI\_INT имеет отображение {(int,0)}.

Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

* Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов MPI\_Type\_contiguous, MPI\_Type\_vector, MPI\_Type\_hvector, MPI\_Type\_indexed, MPI\_Type\_hindexed, MPI\_Type\_struct.
* Новый производный тип регистрируется вызовом функции MPI\_Type\_commit. Только после регистрации новый производный тип можно использовать в коммуникационных подпрограммах и при конструировании других типов. Предопределенные типы MPI считаются зарегистрированными.
* Когда производный тип становится ненужным, он уничтожается функцией MPI\_Type\_free.

Любой тип данных в MPI имеет две характеристики: протяженность и размер, выраженные в байтах:

* *Протяженность типа* определяет, сколько байт переменная данного типа занимает в памяти. Эта величина может быть вычислена как:  
  адрес последней ячейки данных - адрес первой ячейки данных + длина последней ячейки данных (опрашивается подпрограммой MPI\_Type\_extent).
* *Размер типа* определяет количество реально передаваемых байт в коммуникационных операциях. Эта величина равна сумме длин всех базовых элементов определяемого типа (опрашивается подпрограммой MPI\_Type\_size).

Для простых типов протяженность и размер совпадают.

***Функция MPI\_Type\_extent*** определяет протяженность элемента некоторого типа.

C:

int MPI\_Type\_extent(MPI\_Datatype datatype, MPI\_Aint \*extent)

FORTRAN:

MPI\_TYPE\_EXTENT(DATATYPE, EXTENT, IERROR)

INTEGER DATATYPE, EXTENT, IERROR

|  |  |  |
| --- | --- | --- |
| IN | datatype | - тип данных; |
| OUT | extent | - протяженность элемента заданного типа. |

***Функция MPI\_Type\_size*** определяет "чистый" размер элемента некоторого типа (за вычетом пустых промежутков).

C:

int MPI\_Type\_size(MPI\_Datatype datatype, int \*size)

FORTRAN:

MPI\_TYPE\_SIZE(DATATYPE, SIZE, IERROR)

INTEGER DATATYPE, SIZE, IERROR

|  |  |  |
| --- | --- | --- |
| IN | datatype | - тип данных; |
| OUT | size | - размер элемента заданного типа. |

Как отмечалось выше, для создания производных типов в MPI имеется набор специальных функций-конструкторов. Рассмотрим их в последовательности от простого к сложному.

Самый простой ***конструктор типа MPI\_Type\_contiguous*** создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти.

C:

int MPI\_Type\_contiguous(int count, MPI\_Datatype oldtype,

MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

|  |  |  |
| --- | --- | --- |
| IN | count | - число элементов базового типа; |
| IN | oldtype | - базовый тип данных; |
| OUT | newtype | - новый производный тип данных. |

Графическая интерпретация работы конструктора MPI\_Type\_contiguousа приведена на рис. 5.1.

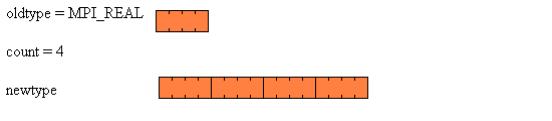


Рис. 5.1. Графическая интерпретация работы конструктораа MPI\_Type\_contiguous.

***Конструктор типа MPI\_Type\_vector*** создает тип, элемент которого представляет собой несколько равноудаленных друг от друга блоков из одинакового числа смежных элементов базового типа.

C:

int MPI\_Type\_vector(int count, int blocklength, int stride,

MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,

NEWTYPE, IERROR)

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | count | - | число блоков; |
| IN | blocklength | - | число элементов базового типа в каждом блоке; |
| IN | stride | - | шаг между началами соседних блоков, измеренный числом элементов базового типа; |
| IN | oldtype | - | базовый тип данных; |
| OUT | newtype | - | новый производный тип данных. |

Функция создает тип newtype, элемент которого состоит из count блоков, каждый из которых содержит одинаковое число blocklength элементов типа oldtype. Шаг stride между началом блока и началом следующего блока всюду одинаков и кратен протяженности представления базового типа. Графическая интерпретация работы конструктора MPI\_Type\_vector приведена на рис. 5.2.

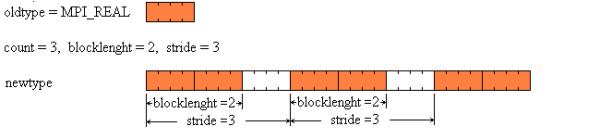


Рис. 5.2. Графическая интерпретация работы конструктора MPI\_Type\_vector.

***Конструктор типа MPI\_Type\_hvector*** расширяет возможности конструктора MPI\_Type\_vector, позволяя задавать произвольный шаг между началами блоков в байтах.

C:

int MPI\_Type\_hvector(int count, int blocklength, MPI\_Aint stride,

MPI\_Datatype oldtype, MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,

NEWTYPE, IERROR)

INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | count | - | число блоков; |
| IN | blocklength | - | число элементов базового типа в каждом блоке; |
| IN | stride | - | шаг между началами соседних блоков в байтах; |
| IN | oldtype | - | базовый тип данных; |
| OUT | newtype | - | новый производный тип данных. |

Графическая интерпретация работы конструктора MPI\_Type\_hvector приведена на рис. 5.3.

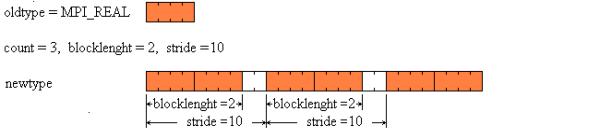


Рис. 5.3. Графическая интерпретация работы конструктора MPI\_Type\_hvector.

***Конструктор типа MPI\_Type\_indexed*** является более универсальным конструктором по сравнению с MPI\_Type\_vector, так как элементы создаваемого типа состоят из произвольных по длине блоков с произвольным смещением блоков от начала размещения элемента. Смещения измеряются в элементах старого типа.

C:

int MPI\_Type\_indexed(int count, int \*array\_of\_blocklengths,

int \*array\_of\_displacements, MPI\_Datatype oldtype,  
MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_INDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS,

ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), ARRAY\_OF\_DISPLACEMENTS(\*),

OLDTYPE, NEWTYPE, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | count | - | число блоков; |
| IN | array\_of\_blocklengths | - | массив, содержащий число элементов базового типа в каждом блоке; |
| IN | array\_of\_displacements | - | массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются числом элементов базового типа; |
| IN | oldtype | - | базовый тип данных; |
| OUT | newtype | - | новый производный тип данных. |

Эта функция создает тип newtype, каждый элемент которого состоит из count блоков, где i-ый блок содержит array\_of\_blocklengths[i] элементов базового типа и смещен от начала размещения элемента нового типа на array\_of\_displacements[i] элементов базового типа. Графическая интерпретация работы конструктора MPI\_Type\_indexedа приведена на рис. 5.4.

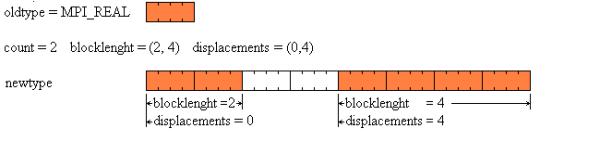


Рис. 5.4. Графическая интерпретация работы конструктора MPI\_Type\_indexed.

***Конструктор типа MPI\_Type\_hindexed*** идентичен конструктору MPI\_Type\_indexed за исключением того, что смещения измеряются в байтах.

C:

int MPI\_Type\_hindexed(int count, int \*array\_of\_blocklengths,

MPI\_Aint \*array\_of\_displacements, MPI\_Datatype oldtype,  
MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_HINDEXED(COUNT, ARRAY\_OF\_BLOCKLENGTHS,

ARRAY\_OF\_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), ARRAY\_OF\_DISPLACEMENTS(\*),

OLDTYPE, NEWTYPE, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | count | - | число блоков; |
| IN | array\_of\_blocklengths | - | массив, содержащий число элементов базового типа в каждом блоке; |
| IN | array\_of\_displacements | - | массив смещений каждого блока от начала размещения элемента нового типа, смещения измеряются в байтах; |
| IN | oldtype | - | базовый тип данных; |
| OUT | newtype | - | новый производный тип данных. |

Элемент нового типа состоит из count блоков, где i-ый блок содержит array\_of\_blocklengths[i] элементов старого типа и смещен от начала размещения элемента нового типа на array\_of\_displacements[i] байт. Графическая интерпретация работы конструктора MPI\_Type\_hindexed приведена на рис. 5.5.

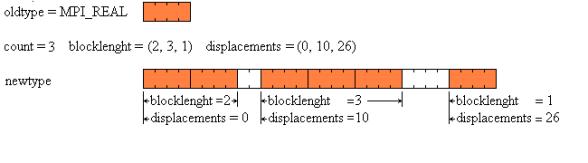


Рис. 5.5. Графическая интерпретация работы конструктора MPI\_Type\_hindexed.

***Конструктор типа MPI\_Type\_struct*** - самый универсальный из всех конструкторов типа. Создаваемый им тип является структурой, состоящей из произвольного числа блоков, каждый из которых может содержать произвольное число элементов одного из базовых типов и может быть смещен на произвольное число байтов от начала размещения структуры.

C:

int MPI\_Type\_struct(int count, int \*array\_of\_blocklengths,

MPI\_Aint \*array\_of\_displacements, MPI\_Datatype \*array\_of\_types,  
MPI\_Datatype \*newtype)

FORTRAN:

MPI\_TYPE\_STRUCT(COUNT, ARRAY\_OF\_BLOCKLENGTHS,

ARRAY\_OF\_DISPLACEMENTS, ARRAY\_OF\_TYPES, NEWTYPE, IERROR)

INTEGER COUNT, ARRAY\_OF\_BLOCKLENGTHS(\*), ARRAY\_OF\_DISPLACEMENTS(\*),

ARRAY\_OF\_TYPES(\*), NEWTYPE, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | count | - | число блоков; |
| IN | array\_of\_blocklength | - | массив, содержащий число элементов одного из базовых типов в каждом блоке; |
| IN | array\_of\_displacements | - | массив смещений каждого блока от начала размещения структуры, смещения измеряются в байтах; |
| IN | array\_of\_type | - | массив, содержащий тип элементов в каждом блоке; |
| OUT | newtype | - | новый производный тип данных. |

Функция создает тип newtype, элемент которого состоит из count блоков, где i-ый блок содержит array\_of\_blocklengths[i] элементов типа array\_of\_types[i]. Смещение i-ого блока от начала размещения элемента нового типа измеряется в байтах и задается в array\_of\_displacements[i].

Графическая интерпретация работы конструктора MPI\_Type\_struct приведена на рис. 5.6.

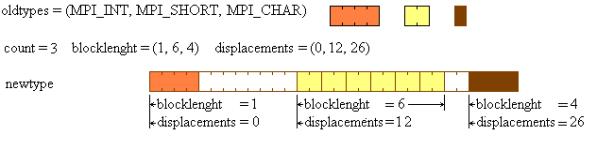


Рис. 5.6. Графическая интерпретация работы конструктора MPI\_Type\_struct.

***Функция MPI\_Type\_commit*** регистрирует созданный производный тип. Только после регистрации новый тип может использоваться в коммуникационных операциях.

C:

int MPI\_Type\_commit(MPI\_Datatype \*datatype)

FORTRAN:

MPI\_TYPE\_COMMIT(DATATYPE, IERROR)

INTEGER DATATYPE, IERROR

INOUT datatype   - новый производный тип данных.

***Функция MPI\_Type\_free*** уничтожает описатель производного типа.

C:

int MPI\_Type\_free(MPI\_Datatype \*datatype)

FORTRAN:

>

MPI\_TYPE\_FREE(DATATYPE, IERROR)

INTEGER DATATYPE, IERROR

INOUT datatype   - уничтожаемый производный тип.

Функция MPI\_Type\_free устанавливает описатель типа в состояние MPI\_DATATYPE\_NULL. Это не повлияет на выполняющиеся в данный момент коммуникационные операции с этим типом данных и на производные типы, которые ранее были определены через уничтоженный тип.

Для определения длины сообщения используются две функции: MPI\_Get\_count и MPI\_Get\_elements. Для сообщений из простых типов они возвращают одинаковое число. Подпрограмма MPI\_Get\_count возвращает число элементов типа datatype, указанного в операции получения. Если получено не целое число элементов, то она возвратит константу MPI\_UNDEFINED (функция MPI\_Get\_count рассматривалась в разделе [3.2.](http://rsusu1.rnd.runnet.ru/tutor/method/m2/page06.html#3.2Get_c), посвященном коммуникационным операциям типа точка-точка).

***Функция MPI\_Get\_elements*** возвращает число элементов простых типов, содержащихся в сообщении.

C:

int MPI\_Get\_elements(MPI\_Status \*status, MPI\_Datatype datatype,

int \*count)

FORTRAN:

MPI\_GET\_ELEMENTS(STATUS, DATATYPE, COUNT, IERROR)

INTEGER STATUS(MPI\_STATUS\_SIZE), DATATYPE, COUNT, IERROR

|  |  |  |  |
| --- | --- | --- | --- |
| IN | status | - | статус сообщения; |
| IN | datatype | - | тип элементов сообщения; |
|  |  |  |  |
| OUT | count | - | число элементов простых типов, содержащихся в сообщении. |
|  |  |  |  |

**MPI\_Type\_size**

Return the number of bytes occupied by entries in the datatype

**int MPI\_Type\_size(**

**MPI\_Datatype** *datatype***,**

**int** \**size*

**);**

**Parameters**

*datatype*

[in] datatype (handle)

*size*

[out] datatype size (integer)

**Remarks**

MPI\_TYPE\_SIZE returns the total size, in bytes, of the entries in the type signature associated with datatype; i.e., the total size of the data in a message that would be created with this datatype. Entries that occur multiple times in the datatype are counted with their multiplicity.

**Thread and Interrupt Safety**

This routine is both thread- and interrupt-safe. This means that this routine may safely be used by multiple threads and from within a signal handler.

**Notes for Fortran**

All MPI routines in Fortran (except for MPI\_WTIME and MPI\_WTICK) have an additional argument ierr at the end of the argument list. ierr is an integer and has the same meaning as the return value of the routine in C. In Fortran, MPI routines are subroutines, and are invoked with the call statement.

All MPI objects (e.g., MPI\_Datatype, MPI\_Comm) are of type INTEGER in Fortran.

**Errors**

All MPI routines (except [MPI\_Wtime](https://mpi.deino.net/mpi_functions/MPI_Wtime.html) and [MPI\_Wtick](https://mpi.deino.net/mpi_functions/MPI_Wtick.html)) return an error value; C routines as the value of the function and Fortran routines in the last argument. Before the value is returned, the current MPI error handler is called. By default, this error handler aborts the MPI job. The error handler may be changed with [MPI\_Comm\_set\_errhandler](https://mpi.deino.net/mpi_functions/MPI_Comm_set_errhandler.html) (for communicators), [MPI\_File\_set\_errhandler](https://mpi.deino.net/mpi_functions/MPI_File_set_errhandler.html) (for files), and [MPI\_Win\_set\_errhandler](https://mpi.deino.net/mpi_functions/MPI_Win_set_errhandler.html) (for RMA windows). The MPI-1 routine [MPI\_Errhandler\_set](https://mpi.deino.net/mpi_functions/MPI_Errhandler_set.html) may be used but its use is deprecated. The predefined error handler MPI\_ERRORS\_RETURN may be used to cause error values to be returned. Note that MPI does *not* guarentee that an MPI program can continue past an error; however, MPI implementations will attempt to continue whenever possible.

*MPI\_SUCCESS*

No error; MPI routine completed successfully.

*MPI\_ERR\_TYPE*

Invalid datatype argument. May be an uncommitted MPI\_Datatype (see [MPI\_Type\_commit](https://mpi.deino.net/mpi_functions/MPI_Type_commit.html)).

*MPI\_ERR\_ARG*

Invalid argument. Some argument is invalid and is not identified by a specific error class (e.g., MPI\_ERR\_RANK).

**Example Code**

The following sample code illustrates [MPI\_Type\_size](https://mpi.deino.net/mpi_functions/MPI_Type_size.html).

#include "mpi.h"  
#include <stdio.h>  
#include <string.h>  
   
int main(int argc, char \*\*argv)  
{  
    int mpi\_err, errs = 0, size;  
    MPI\_Aint lb, ub, extent;  
    MPI\_Datatype type;  
    struct { float a; int b; } foo;  
  
    [MPI\_Init](https://mpi.deino.net/mpi_functions/MPI_Init.html)(&argc, &argv);  
   
    type = MPI\_INT;  
    [MPI\_Type\_size](https://mpi.deino.net/mpi_functions/MPI_Type_size.html)(type, &size);  
    if (size != sizeof(int)) {  
        fprintf(stderr, "[MPI\_Type\_size](https://mpi.deino.net/mpi_functions/MPI_Type_size.html) of MPI\_INT incorrect size (%d); should be %d.\n", size, (int) sizeof(int));fflush(stderr);  
        errs++;  
    }  
   
    [MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html)(type, &lb, &extent);  
    if (extent != sizeof(int)) {  
        fprintf(stderr, "[MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html) of MPI\_INT returned incorrect extent (%d); should be %d.\n", (int) extent, (int) sizeof(int));fflush(stderr);  
        errs++;  
    }  
    if (lb != 0) {  
        fprintf(stderr, "[MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html) of MPI\_INT returned incorrect lb (%d); should be 0.\n", (int) lb);fflush(stderr);  
        errs++;  
    }  
   
    [MPI\_Type\_ub](https://mpi.deino.net/mpi_functions/MPI_Type_ub.html)(type, &ub);  
    if (ub != extent - lb) {  
        fprintf(stderr, "[MPI\_Type\_ub](https://mpi.deino.net/mpi_functions/MPI_Type_ub.html) of MPI\_INT returned incorrect ub (%d); should be %d.\n", (int) ub, (int) (extent - lb));fflush(stderr);  
        errs++;  
    }  
   
    type = MPI\_FLOAT\_INT;  
    [MPI\_Type\_size](https://mpi.deino.net/mpi_functions/MPI_Type_size.html)(type, &size);  
    if (size != sizeof(float) + sizeof(int)) {  
        fprintf(stderr, "[MPI\_Type\_size](https://mpi.deino.net/mpi_functions/MPI_Type_size.html) of MPI\_FLOAT\_INT returned incorrect size (%d); should be %d.\n", size, (int) (sizeof(float) + sizeof(int)));fflush(stderr);  
        errs++;  
    }  
   
    [MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html)(type, &lb, &extent);  
    if (extent != sizeof(foo)) {  
        fprintf(stderr, "[MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html) of MPI\_FLOAT\_INT returned incorrect extent (%d); should be %d.\n", (int) extent, (int) sizeof(foo));fflush(stderr);  
        errs++;  
    }  
    if (lb != 0) {  
        fprintf(stderr, "[MPI\_Type\_get\_extent](https://mpi.deino.net/mpi_functions/MPI_Type_get_extent.html) of MPI\_FLOAT\_INT returned incorrect lb (%d); should be 0.\n", (int) lb);fflush(stderr);  
        errs++;  
    }  
   
    [MPI\_Type\_ub](https://mpi.deino.net/mpi_functions/MPI_Type_ub.html)(type, &ub);  
    if (ub != extent - lb) {  
        fprintf(stderr, "[MPI\_Type\_ub](https://mpi.deino.net/mpi_functions/MPI_Type_ub.html) of MPI\_FLOAT\_INT returned incorrect ub (%d); should be %d.\n", (int) ub, (int) (extent - lb));fflush(stderr);  
        errs++;  
    }  
   
    [MPI\_Finalize](https://mpi.deino.net/mpi_functions/MPI_Finalize.html)();  
    return errs;  
}

In C, [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) and [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) are distinct:

* [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) is the type of a variable able to contain a memory address. It is used in heterogeneous datatype creation routines for instance, such as [MPI\_Type\_create\_hindexed](https://www.rookiehpc.com/mpi/docs/mpi_type_create_hindexed.php), [MPI\_Type\_create\_hindexed\_block](https://www.rookiehpc.com/mpi/docs/mpi_type_create_hindexed_block.php), [MPI\_Type\_create\_hvector](https://www.rookiehpc.com/mpi/docs/mpi_type_create_hvector.php) and [MPI\_Type\_create\_struct](https://www.rookiehpc.com/mpi/docs/mpi_type_create_struct.php). Please see [MPI\_ADDRESS\_KIND](https://www.rookiehpc.com/mpi/docs/mpi_address_kind.php) for the FORTRAN counterpart.
* [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) is an [MPI\_Datatype](https://www.rookiehpc.com/mpi/docs/mpi_datatype.php) used to inform MPI about the type of a variable passed to a routine. Similar to [MPI\_INT](https://www.rookiehpc.com/mpi/docs/mpi_int.php) being the [MPI\_Datatype](https://www.rookiehpc.com/mpi/docs/mpi_datatype.php) corresponding to an int, [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) is the [MPI\_Datatype](https://www.rookiehpc.com/mpi/docs/mpi_datatype.php) corresponding to an [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php).
* #include <stdio.h>
* #include <stdlib.h>
* #include <mpi.h>
* /\*\*
* \* @brief Illustrate how to use the [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) type and the [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) [MPI\_Datatype](https://www.rookiehpc.com/mpi/docs/mpi_datatype.php).
* \* @details This application consists of 2 MPI processes. The MPI process 0
* \* sends the address takes the address of its local variable to the MPI process
* \* 1, which prints it.
* \*\*/
* **int** main(**int** argc, **char**\* argv[])
* {
* [MPI\_Init](https://www.rookiehpc.com/mpi/docs/mpi_init.php)(&argc, &argv);
* // Check that only 2 MPI Processes are used
* **int** comm\_size;
* [MPI\_Comm\_size](https://www.rookiehpc.com/mpi/docs/mpi_comm_size.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), &comm\_size);
* **if**(comm\_size != 2)
* {
* printf("This application is meant to be run with 2 MPI processes, not %d.\n", comm\_size);
* [MPI\_Abort](https://www.rookiehpc.com/mpi/docs/mpi_abort.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), EXIT\_FAILURE);
* }
* **int** my\_rank;
* [MPI\_Comm\_rank](https://www.rookiehpc.com/mpi/docs/mpi_comm_rank.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), &my\_rank);
* **enum** roles {SENDER, RECEIVER};
* **switch**(my\_rank)
* {
* **case** SENDER:
* {
* // Declare a local variable, get its address and send it to MPI Process 0
* **int** my\_variable;
* [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) my\_variable\_address;
* [MPI\_Get\_address](https://www.rookiehpc.com/mpi/docs/mpi_get_address.php)(&my\_variable, &my\_variable\_address);
* printf("[MPI Procees %d] The address of my local variable is %p.\n", my\_rank, (**void**\*)my\_variable\_address);
* [MPI\_Send](https://www.rookiehpc.com/mpi/docs/mpi_send.php)(&my\_variable\_address, 1, [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php), 1, 0, [MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php));
* **break**;
* }
* **case** RECEIVER:
* {
* // Get the address of the local variable held on MPI process 1 and print it
* [MPI\_Aint](https://www.rookiehpc.com/mpi/docs/mpi_aint.php) remote\_variable\_address;
* [MPI\_Recv](https://www.rookiehpc.com/mpi/docs/mpi_recv.php)(&remote\_variable\_address, 1, [MPI\_AINT](https://www.rookiehpc.com/mpi/docs/mpi_aint.php), 0, 0, [MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), [MPI\_STATUS\_IGNORE](https://www.rookiehpc.com/mpi/docs/mpi_status_ignore.php));
* printf("[MPI Procees %d] The address of the remote variable is %p.\n", my\_rank, (**void**\*)remote\_variable\_address);
* **break**;
* }
* }
* [MPI\_Finalize](https://www.rookiehpc.com/mpi/docs/mpi_finalize.php)();
* **return** EXIT\_SUCCESS;
* }