Derived datatypes

? Questions

- How can you reduce the number of messages sent and received?
- How can you use your own derived datatypes as content of messages?

Objectives

- Understand how MPI handles datatypes.
- Learn to send and receive messages using composite datatypes.
- Learn how to represent homogeneous collections as MPI datatypes.
- Learn how to represent your own derived datatypes as MPI datatypes.

The ability to define custom datatypes is one of the hallmarks of a modern programming language, since it allows programmers to structure their code in a way that enhances readability and maintainability. How can this be done in MPI? Recall that MPI is a standard describing a library to enable parallel programming in the message passing model.

MPI supports many of the basic datatypes recognized by the C and Fortran standards.

Basic datatypes in MPI and in the C standard. For a comprehensive explanation of the types defined in the C language, you can consult this reference.

MPI	C
`MPI_CHAR`	`signed char`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_WCHAR`	`wchar_t`
`MPI_SHORT`	`short`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_LONG_LONG_INT`	`long long`
`MPI_SIGNED_CHAR`	`signed char`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_UNSIGNED_LONG_LONG`	`unsigned long long`
`MPI_C_COMPLEX`	`float _Complex`
`MPI_C_DOUBLE_COMPLEX`	`double _Complex`
`MPI_C_LONG_DOUBLE_COMPLEX`	`long double _Complex`
`MPI_PACKED`	
`MPI_BYTE`	

In the C language, types are **primitive** constructs: they are *defined* by the standard and *enforced* by the compiler. The MPI types are instead **variants** in the `MPI_Datatype` enumeration: they appear as the **same** type to the compiler. This is a fundamental difference which influences the way custom datatypes are handled.

In the C language, you would declare a `struct` such as the following:

```
struct Pair {
  int first;
  char second;
};
```

`Pair` is a new type. From the compiler's point of view, it has status on par with the fundamental datatypes introduced above. The C standard makes requirements on *how to* represent this in memory and the compiler will generate machine code to comply with it.

MPI does not know how to represent user-defined datatypes in memory by itself:

- How much memory does it need? Recall that MPI deals with **groups of processes**. For portability, you can *never* assume that two processes share the same architecture!
- How are the components of `Pair` laid out in memory? Are they always contiguous? Or are they padded?

The programmer needs to provide this low-level information, such that the MPI runtime can send and receive custom datatypes as messages over a heterogeneous network of processes.

Representation of datatypes in MPI

The representation of datatypes in MPI uses few low-level concepts. The **type signature** of a custom datatypes is the list of its basic datatypes:

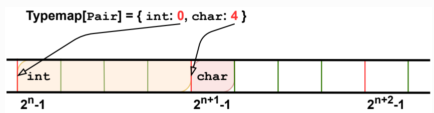
$$\text{Type signature}[\mathtt{T}] = [\mathtt{Datatype}_0, \ldots, \mathtt{Datatype}_{n-1}] \tag{1}$$

The **typemap** is the associative array (map) with datatypes, as understood by MPI, as *keys* and displacements, in bytes, as *values*.

$$\text{Typemap}[\mathtt{T}] = \{\mathtt{Datatype}_0 : \text{Displacement}_0, \ldots, \mathtt{Datatype}_{n-1} : \text{Displacement}_{n-1}\} \tag{2}$$

The displacements are *relative* to the buffer the datatype describes.

Assuming that an `int` takes 4 bytes of memory, the typemap for our `Pair` datatype would be: $\text{Typemap}[\mathtt{Pair}] = \{\mathtt{int} : 0, \mathtt{char} : 4\}$. Note again that the displacements are *relative*.



Depiction of the typemap for the `Pair` custom type. The displacements are always relative.

Knowledge of typemap and type signature is not enough for a full description of the type to the MPI runtime: the underlying programming language might mandate architecture-specific alignment of the basic datatypes. The data structure would then be laid out in memory incoherently with the displacements in its typemap. We need a few more concepts. Given a typemap m we can define:

Lower bound

The first byte occupied by the datatype.

$$\text{LB}[m] = \min_j [\text{Displacement}_j] \tag{3}$$

Upper bound

The last byte occupied by the datatype.

$$\text{UB}[m] = \max_j [\text{Displacement}_j + \mathtt{sizeof}(\text{Datatype}_j)] + \text{Padding} \tag{4}$$

Extent

The amount of memory needed to represent the datatype, taking into account architecture-specific alignment.

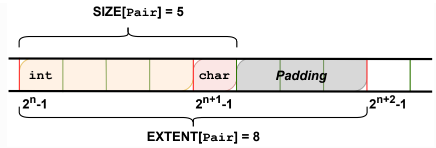
$$\text{Extent}[m] = \text{UB}[m] - \text{LB}[m] \tag{5}$$

The C language (and Fortran) *require* that the data occurs in memory at well-defined addresses: the data needs to be aligned. The address, in bytes, of any item must be a multiple of the size of that item in bytes. This is so-called *natural alignment*. For our `Pair` data structure the `first` element is an `int` and occupies 4 bytes. An `int` will align to 4 bytes boundaries: when allocating a new `int` in memory, the compiler will insert **padding** to reach the alignment boundary. Indeed, `second` is a `char` and requires just 1 byte. This gives:

$$\begin{aligned} \mathtt{Pair.first} &\rightarrow \text{Displacement}_0 = 0, \quad \mathtt{sizeof}(\mathtt{int}) = 4 \\ \mathtt{Pair.second} &\rightarrow \text{Displacement}_1 = 4, \quad \mathtt{sizeof}(\mathtt{char}) = 1 \end{aligned}$$

To insert yet another `Pair` item, we first need to reach the alignment boundary with a padding of 3 bytes. Thus:

$$\begin{aligned} \text{LB}[\mathtt{Pair}] &= \min[0, 4] = 0 \\ \text{UB}[\mathtt{Pair}] &= \max[0 + 4, 4 + 1] + 3 = 8 \\ \text{Extent}[\mathtt{Pair}] &= \text{UB}[\mathtt{Pair}] - \text{LB}[\mathtt{Pair}] = 8 \end{aligned}$$



*The relation between **size** and **extent** of a derived datatype in the case of the `Pair`. We show the address alignment boundaries with vertical red lines. The lowerbound of the custom datatype is 4: `first` can be found with an offset of 4 bytes after the starting address. Notice the 3 bytes of padding, necessary to achieve natural alignment of `Pair`. The upperbound is 8: the next item of type `Pair` can be found with an offset of 8 bytes after the previous element. The total size is 5 bytes, but the extent, which takes the padding into account, is 8 bytes.*

✍ Which of the following statements about the size and extent of an MPI datatype is true?

1. The size is always greater than the extent
2. The size and extent can be equal
3. The extent is always greater than the size
4. None of the above

✔ Solution

Click to show

MPI offers functions to query extent and size of its types: they all take a variant of the `MPI_Datatype` enumeration as argument.

`MPI_Type_get_extent`

Returns the lower bound and extent of a type.

```
int MPI_Type_get_extent(MPI_Datatype type,
                        MPI_Aint *lb,
                        MPI_Aint *extent)
```

💻 Parameters

Click to show

`MPI_Type_size`

Returns the number of bytes occupied by entries in the datatype.

```
int MPI_Type_size(MPI_Datatype type,
                  int *size)
```

💻 Parameters

Click to show

✍ Extents and sizes

We will now play around a bit with the compiler and MPI to gain further understanding of padding, alignment, extents, and sizes.

1. What are extents and sizes for the basis datatypes `char`, `int`, `float`, and `double` on your architecture? Do the numbers conform to your expectations? What is the result of `sizeof` for these types?

   ```
   // char
   printf("sizeof(char) = %ld\n", sizeof(char));
   MPI_Type_get_extent(MPI_CHAR, &.., &..);
   MPI_Type_size(MPI_CHAR, &..);
   printf("For MPI_CHAR:\n  lowerbound = %ld; extent = %ld; size = %d\n", ..,
           .., ..);
   ```

   Download a working solution

2. Let's now look at the `Pair` data structure. We first need declare the data structure to MPI. The following code, which we will study in much detail later on, achieves the purpose:

   ```
   // build up the typemap for Pair
   // the type signature for Pair
   MPI_Datatype typesig[2] = {MPI_INT, MPI_CHAR};
   // how many of each type in a "block" of Pair
   int block_lengths[2] = {1, 1};
   // displacements of data members in Pair
   MPI_Aint displacements[2];
   // why not use pointer arithmetic directly?
   MPI_Get_address(&my_pair.first, &displacements[0]);
   MPI_Get_address(&my_pair.second, &displacements[1]);

   // create and commit the new type
   ```

```
MPI_Datatype mpi_pair;
MPI_Type_create_struct(2, block_lengths, displacements, typesig, &mpi_pair);
MPI_Type_commit(&mpi_pair);
```

What are the size and the extent? Do they match up with our pen-and-paper calculation? Try different combinations of datatypes and adding other fields to the `struct`.

Download a working solution

Extents and the `count` parameter

Let us reiterate: the extent of a custom datatype *is not* its size. The extent tells the MPI runtime how to get to the **next** item in an array of a given type, much like a *stride*.

We can send an array of `n` `int`-s with a single `MPI_Send`:

```
if (rank == 0) {
  fprintf(stdout, "rank %d send\n", rank);
  for (int i = 0; i < SIZE; ++i) {
    fprintf(stdout, "buffer[%d] = %d\n", i, buffer[i]);
  }
  MPI_Send(buffer, SIZE, MPI_INT, 1, 0, comm);
} else {
  MPI_Recv(buffer, SIZE, MPI_INT, 0, 0, comm, &status);
  fprintf(stdout, "rank %d recv\n", rank);
  for (int i = 0; i < SIZE; ++i) {
    fprintf(stdout, "buffer[%d] = %d\n", i, buffer[i]);
  }
}
```

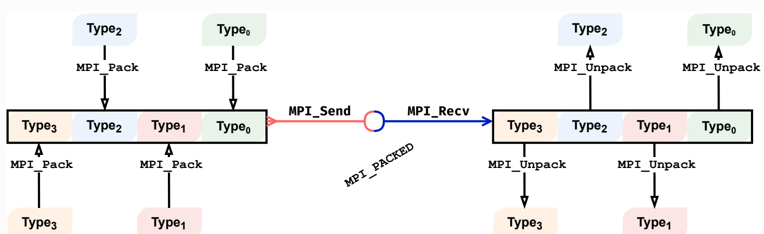
or with `n` such calls:

```
if (rank == 0) {
  for (int i = 0; i < SIZE; ++i) {
    fprintf(stdout, "rank %d send: buffer[%d] = %d\n", rank, i, buffer[i]);
    MPI_Send(buffer + (i * extent), 1, MPI_INT, 1, 0, comm);
  }
} else {
  for (int i = 0; i < SIZE; ++i) {
    MPI_Recv(buffer + (i * extent), 1, MPI_INT, 0, 0, comm, &status);
    fprintf(stdout, "rank %d recv: buffer[%d] = %d\n", rank, i,
            buffer[i]);
  }
}
```

In the latter case, we must program explicitly how to get the next element in the array by using the extent of the datatype.

Packing and unpacking

MPI offers the possibility to pack and unpack data of known datatype into a single contiguous memory buffer, *without* first having to define a corresponding datatype. This can be an extremely useful technique to reduce messaging traffic and could help with the readability and portability of the code. The resulting packed buffer will be of type `MPI_PACKED` and can contain any sort of heterogeneous collection of basic datatypes recognized by MPI.

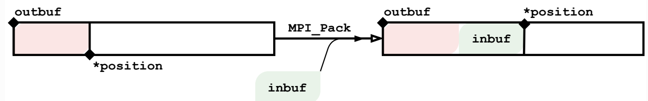


MPI allows the programmer to communicate heterogeneous collections into a single message, without defining a full-fledged custom datatype. The data is packed into a buffer of type `MPI_PACKED`. On the receiving end, the buffer will be unpacked into its constituent components.

`MPI_Pack`

Pack data in noncontiguous memory to a contiguous memory buffer.

```
int MPI_Pack(const void *inbuf,
             int incount,
             MPI_Datatype datatype,
             void *outbuf,
             int outsize,
             int *position,
             MPI_Comm comm)
```



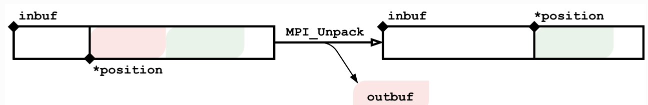
The relation of `inbuf`, `outbuf`, and `position` when calling `MPI_Pack`. In this figure, `outbuf` already holds some data (the red shaded area). The data in `inbuf` is copied to `outbuf` starting at the address `outbuf+*position`. When the function returns, the `position` parameter will have been updated to refer to the first position in `outbuf` following the data copied by this call.

💻 Parameters Click to show

`MPI_Unpack`

Unpack a contiguous memory buffer into noncontiguous memory locations.

```
int MPI_Unpack(const void *inbuf,
               int insize,
               int *position,
               void *outbuf,
               int outcount,
               MPI_Datatype datatype,
               MPI_Comm comm)
```



The relation of `inbuf`, `outbuf`, and `position` when calling `MPI_Unpack`. In this figure, `inbuf` holds some data. The data in `inbuf` is copied to `outbuf` starting at the address given with `position`. When the function returns, the `position` parameter will have been updated to the first position in `inbuf` following the just copied data.

💻 Parameters Click to show

✍️ Message passing Pokémons

In the Pokémon trading card game, opponents face each in duels using their pokémons. The game is played in turns and at each turn a player can attack. We have to send:

- The attacking pokémon's name: a `char` array.
- How many life points it has: a `double`.
- The damage its attack will inflict: an `int`.
- A damage multiplier: a `double`.

Pack and unpack | Bonus | Superbonus

1. Download the scaffold source code. Open it and read through it.
2. Pack the data in the `message` buffer.
3. Unpack the `message` buffer into its component data.

Compile with:

```
mpicc -g -Wall -std=c11 pokemon-pack-unpack.c -o pokemon-pack-unpack
```

- Why are we hardcoding the length of the pokémon's name?
- What is the purpose of the `position` variable? Print its value after each packing and unpacking. Do these values conform with your intuition?

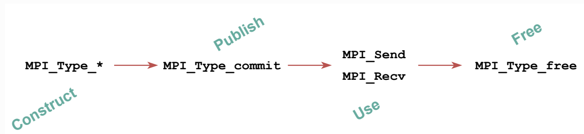
Download a working solution

- Should packing and unpacking happen in the same order? What happens if not?
- What happens when there is a mismatch of types between packing and unpacking?
- We could have packed our data as `char`, `int`, `double`, and `double`. Is there a way to pack (unpack) the life points and the damage multiplier with one call to `MPI_Pack` (`MPI_Unpack`)?

Any type you like: datatype constructors in MPI

The typemap concept allows us to provide a *low-level* description of any compound datatype. The class of functions `MPI_Type_*` offers facilities for *portable* type manipulations in the MPI standard. At a glance, each custom datatype goes through a well-defined lifecycle in an MPI application:

- We *construct* our new datatype with a **type constructor**. The new type will be a variable with `MPI_Datatype` type.
- We *publish* our new type to the runtime with `MPI_Type_commit`.
- We *use* the new type in any of the MPI communication routines, as needed.
- We *free* the new type from memory with `MPI_Type_free`.



The lifecycle of user-defined datatypes in MPI. Calling any of the type constructors will create an object of type `MPI_Datatype` with the user-defined typemap. Before using this custom datatype in message passing, it needs to be published with `MPI_Type_commit`: the typemap is made known to the runtime, allowing it to handle messages of the new custom type. The programmer must take care to free the custom datatype object.

It is not always necessary to go all the way down to a typemap to construct new datatypes in MPI. The following types can be created with convenience functions, side-stepping the explicit computation of a typemap. In MPI nomenclature, these types are:

Contiguous

A homogeneous collection of a given datatype. The returned new type will describe a collection of `count` times the old type. Elements are contiguous: n and $n-1$ are separated by the extent of the old type.

`MPI_Type_contiguous`

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

Vector

A slight generalization of the contiguous type: `count` elements in the new type can be separated by a stride that is an arbitrary multiple of the extent of the old type.

`MPI_Type_vector`

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

Hvector

Yet another generalization of the contiguous datatype. The separation between elements in a hvector is expressed in bytes, rather than as a multiple of the extent.

`MPI_Type_create_hvector`

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

Indexed

This type allows to have non-homogeneous separations between the elements. Each displacement is intended as a multiple of the extent of the old type.

`MPI_Type_indexed`

```
int MPI_Type_indexed(int count,
                     const int array_of_blocklengths[],
                     const int array_of_displacements[],
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype)
```

Hindexed

This is a generalization of the indexed type analogous to the hvector. The non-homogeneous separations between the elements are expressed in bytes, rather than as multiples of the extent.

`MPI_Type_create_hindexed`

```
int MPI_Type_create_hindexed(int count,
                             const int array_of_blocklengths[],
                             const MPI_Aint array_of_displacements[],
                             MPI_Datatype oldtype,
                             MPI_Datatype *newtype)
```

Before using the output parameter `newtype`, it needs to be "published" to the runtime with `MPI_Type_commit`:

`MPI_Type_commit`

```
int MPI_Type_commit(MPI_Datatype *type)
```

`newtype` is a variable of type `MPI_Datatype`. The programmer must ensure proper release of the memory used at the end of the program by calling `MPI_Type_free`:

`MPI_Type_free`

```
int MPI_Type_free(MPI_Datatype *type)
```

In practice, none of the previous convenience constructors might be suitable for your application. As we glimpsed in a previous challenge, the general type constructor `MPI_Type_create_struct` will suit your needs:

`MPI_Type_create_struct`

```
int MPI_Type_create_struct(int count,
                           const int array_of_block_lengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[],
                           MPI_Datatype *newtype)
```

Parameters Click to show

The MPI version of the `Pair` datatype

We saw code for this earlier on, but without explanation. Let's dive into it now!

`Pair` has two fields, hence `count = 2` in the call to `MPI_Type_create_struct`. All array arguments to this function will have length 2. The type signature is:

```
MPI_Datatype typesig[2] = {MPI_INT, MPI_CHAR};
```

We have one `int` in the `first` field and one `char` in the `second` fields, hence the `array_of_block_lengths` argument is:

```
int block_lengths[2] = {1, 1};
```

The calculation of displacements is slightly more involved. We will use `MPI_Get_address` to fill the `displacements` array. Notice that its elements are of type `MPI_Aint`:

```
MPI_Aint displacements[2];
MPI_Get_address(&my_pair.first, &displacements[0]);
MPI_Get_address(&my_pair.second, &displacements[1]);
```

We *cannot use* pointer arithmetic to compute displacements. Always keep in mind that your program might be deployed on heterogeneous architectures: you have to program for correctness and portability.

We are now ready to call the type constructor and commit our type:

```
MPI_Datatype mpi_pair;
MPI_Type_create_struct(2, block_lengths, displacements, typesig, &mpi_pair);
MPI_Type_commit(&mpi_pair);
```

And clean up after use, of course!

```
MPI_Type_free(&mpi_pair);
```

Download the complete source code

✍ More message passing Pokémons

We will revisit the Pokémon example from above using custom datatypes.

Pokémons, again! | Superbonus

1. Download the scaffold source code. Open it and read through it.
2. Define the C `struct` for a pokémon. This has to contain:
   - The attacking pokémon's name: a `char` array.
   - How many life points it has: a `double`.
   - The damage its attack will inflict: an `int`.
   - A damage multiplier: a `double`.
3. Create its corresponding MPI datatype.
4. Print it out on the receiving process.

Compile with:

```
mpicc -g -Wall -std=c11 pokemon-type-create-struct.c -o pokemon-type-create-struct
```

What happens if you don't commit the type?

Download a working solution

See also

- The lecture covering MPI datatypes from EPCC is available on GitHub
- Chapter 5 of the **Using MPI** book by William Gropp *et al*. [GLS14]
- Chapter 6 of the **Parallel Programming with MPI** book by Peter Pacheco. [Pac97]

Keypoints

- A low-level representation as typemap can be associated with any derived data structure.
- Typemaps are essential to enable MPI communication of complex datatypes.
- You can reduce message traffic by packing (unpacking) heterogeneous data together.
- MPI offers many type constructors to portably use your own datatypes in message passing.
- Packing/unpacking are straightforward to use, but might lead to less readable programs.
- Usage of the type constructors can be quite involved, but you strictly ensure your programs will be portable.