COMS3008A: Parallel Computing
Introduction to MPI III

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

2021-10-21

WITS UNIVERSITY

Contents

Outline

MPI Collective Communication cont.

Scatters a buffer in parts to all processes in a communicator, which allows different amounts of data to be sent to different processes.

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

- `sendbuf`: address of send buffer (significant only at root)
- `sendcounts`: integer array (of length group size) specifying the number of elements to send to each processor
- `displs`: integer array (of length group size). Entry i specifies the displacement (relative to `sendbuf` from which to take the outgoing data to process i
- `sendtype`: data type of send buffer elements
- `recvcount`: number of elements in receive buffer (integer)
- `recvtype`: data type of receive buffer elements
- `root`: rank of sending process (integer)

MPI Collective Communication cont.

Example 1

Given an $N \times N$ matrix, A, of integers, write an MPI program that distributes the first M rows of the upper triangle of A to M processes by rows, where each process gets one row of the upper triangle of A (when $M = N$, it means each process gets one row of the upper triangle of A).

For this example, we can use `MPI_Scatterv`.

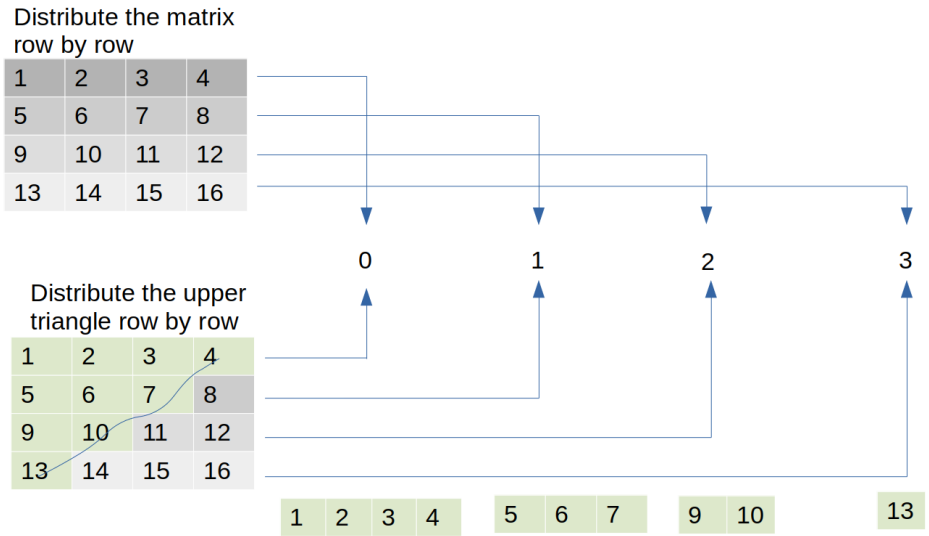


Figure: `MPI_Scatter` (top) and `MPI_Scatterv` (bottom) example

MPI Collective Communication cont.

For Example 1, assuming the matrix is only 4 $\times$ 4, and we are running the MPI code using 4 processes, then some of the arguments of calling `MPI_Scatterv`:

- `sendcounts[4] = {4, 3, 2, 1};`
- `displs[4] = {0, 4, 8, 12}` which is with reference to `sendbuf`; these values can be expressed as `N * rank`, where `rank` is the rank of a process.
- note also that `recvcount` in `MPI_Scatterv` is a scalar; for process 0, `recvcount = 4 (=4-0)`; for process 1, `recvcount = 3 (=4-1)`; for process 2, `recvcount = 2 (=4-2)`; and for process 3, `recvcount = 1 (=4-3)`; so this value can be obtained as `N - rank` where `N` is the number of rows in the matrix, and `rank` is the rank of a process.

`scatterv_1.c` gives an example code for Example 1.

WITS UNIVERSITY

MPI Collective Communication cont.

Sends data from all to all processes; each process may send a different amount of data and provide displacements for the input and output data.

```
MPI_Alltoallv(void *sendbuf, int *sendcounts, int *sdispls,
    MPI_Datatype sendtype, void *recvbuf, int *recvcounts,
    int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
```

- `sendbuf`: starting address of send buffer
- `sendcounts`: integer array equal to the group size specifying the number of elements to send to each processor
- `sdispls`: integer array (of length group size). Entry j specifies the displacement (relative to `sendbuf` from which to take the outgoing data destined for process j
- `sendtype`: data type of send buffer elements
- `recvcounts`: integer array equal to the group size specifying the maximum number of elements that can be received from each processor
- `rdispls`: integer array (of length group size). Entry i specifies the displacement (relative to `recvbuf` at which to place the incoming data from process i
- `recvtype`: data type of receive buffer elements

MPI Collective Communication cont.

Example 2

Given the `MPI_Alltoallv` argument settings shown in the figure (the number of processes is 3), what is the content of `recvbuf` for each process?

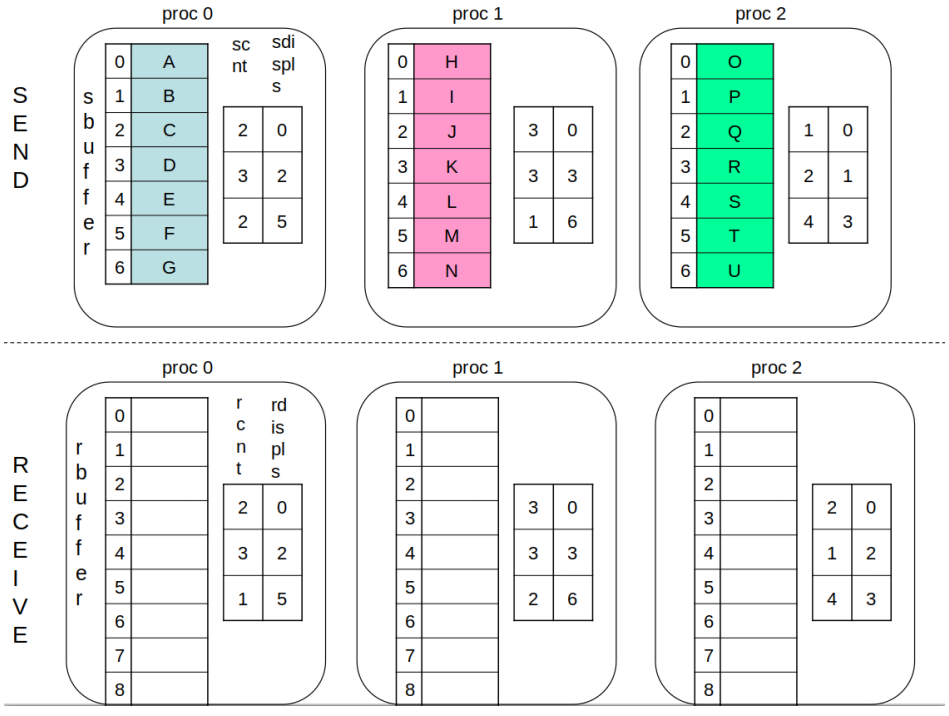


Figure: `MPI_Alltoallv` example

WITS UNIVERSITY

MPI Collective Communication cont.

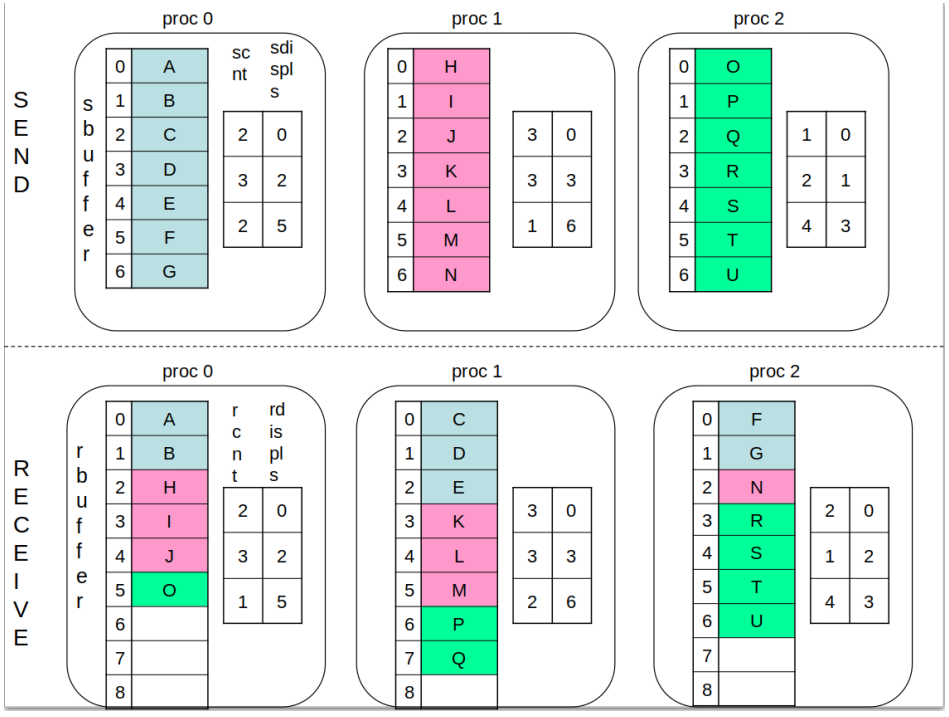


Figure: `MPI_Alltoallv` example

WITS UNIVERSITY

MPI Collective Communication cont.

The following function allows a different number of data elements to be sent by each process by replacing `recvcount` in `MPI_Gather` with an array `recvcounts`

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

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)
- `target`: rank of receiving process (integer)

MPI Collective Communication cont.

Gather data from all processes and deliver the combined data to all processes

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

- `sendbuf`: pointer, starting address of send buffer (or the data to be sent)
- `sendcount`: the number of elements in the send buffer
- `sendtype`: datatype of send buffer elements
- `recvbuf`: pointer, starting address of receive buffer (significant only at root)
- `recvcounts`: integer array (of length group size) containing the number of elements to be received from each process (significant only at root)
- `displs`: integer array (of length group size). Entry i specifies the displacement relative to `recvbuf` at which to place the incoming data from process i (significant only at root)
- `recvtype`: the datatype of data to be received (significant only at root)

Outline

MPI Derived Datatypes

Example 3

```
1    double x[1000];
2    ....
3    for(i=0; i<1000; i++){
4      if(my_rank == 0)
5            MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
6    else
7            MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0,comm,&
                 status);
8    }
9  /*the following is more efficient than using the for
       loop*/
10   if(my_rank == 0)
11       MPI_Send(&x[0], 1000, MPI_DOUBLE, 1, 0, comm);
12   else
13       MPI_Recv(&x[0], 1000, MPI_DOUBLE, 0, 0, comm, &
             status);
```

In distributed-memory systems, communication can be much more expensive than local computation. Thus, if we can reduce the number of communications, we are likely to improve the performance of our programs.

MPI Built-in Datatypes

- The MPI standard defines many built in datatypes, mostly mirroring standard C/C++ or FORTRAN datatypes
- These are sufficient when sending single instances of each type
- They are also usually sufficient when sending contiguous blocks of a single type
- Sometimes, however, we want to send non-contiguous data or data that is comprised of multiple types
- MPI provides a mechanism to create derived datatypes that are built from simple datatypes

MPI Derived Datatypes Contd.

- In MPI, a *derived datatype* can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- Why use derived datatypes?
  - Primitive datatypes are contiguous;
  - Derived datatypes allow you to specify non-contiguous data in a convenient manner and treat it as though it is contiguous;
  - Useful to
    - Make code more readable
    - Reduce number of messages and increase their size (faster since less latency);
    - Make code more efficient if messages of the same size are repeatedly used.

Outline

Typemap

Formally, a derived datatype in MPI is described by a typemap consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. That is,

- a sequence of basic datatypes: $\{type_0, ..., type_{n-1}\}$
- a sequence of integer displacements: $\{displ_0, ..., displ_{n-1}\}$.
- Typemap = $\{(type_0, disp_0), \cdots, (type_{n-1}, disp_{n-1})\}$

For example, a typemap might consist of (double,0),(char,8) indicating the type has two elements:

- a double precision floating point value starting at displacement 0,
- and a single character starting at displacement 8.

Typemap Contd.

- Types also have extent, which indicates how much space is required for the type
- The extent of a type may be more than the sum of the bytes required for each component
- For example, on a machine that requires double-precision numbers to start on an 8-byte boundary, the type (double,0),(char,8) will have an extent of 16 even though it only requires 9 bytes

Outline

Creating and Using a New Datatype

Three steps are necessary to create and use a new datatype in MPI:

- Create the type using one of MPI's type construction functions
- Commit the type using `MPI_Type_commit()`.
- Release the datatype using `MPI_Type_free()` when it is not needed any more.

MPI Derived Datatypes Contd.

MPI provides several methods for constructing derived datatypes to handle a wide variety of situations.

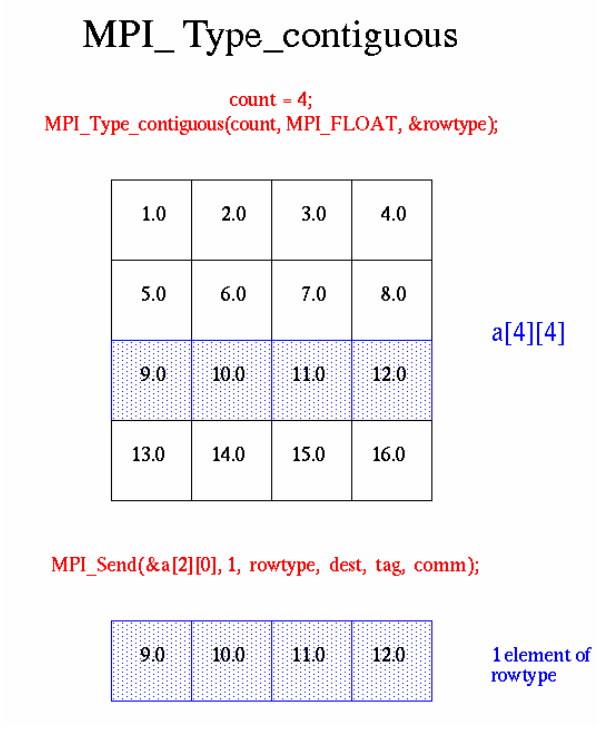
- Contiguous
- Vector
- Indexed
- Struct

Outline

Contiguous Type

Contiguous: The contiguous datatype allows for a single type to refer to contiguous multiple elements of an existing datatype.

```
int MPI_Type_contiguous(
    int count,          //count
    MPI_Datatype oldtype,  //old datatype
    MPI_Datatype *newtype) //new datatype
```



To define the new datatype in this example and release it after finished using it:

```
MPI_Datatype rowtype;
MPI_Type_contiguous(4, MPI_DOUBLE,
                    &rowtype);
MPI_Type_commit(&rowtype);
......
MPI_Type_free(&rowtype);
```

WITS UNIVERSITY

Contiguous Type Contd.

To define a new datatype:

- Declare the new datatype as `MPI_Datatype`.
- Construct the new datatype.
- Before we can use a derived datatype in a communication function, we must first **commit** it with a call to

  ```
  int MPI_Type_commit(MPI_Datatype* datatype);
  ```

  Commits new datatype to the system. Required for all derived datatypes.
- When we finish using the new datatype, we can free any additional storage used with a call to

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

Contiguous Type Contd.

The new datatype is essentially an array of `count` elements having type `oldtype`. For example, the following two code fragments are equivalent:

```
MPI_Send (a,n,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);
```

and

```
MPI_Datatype rowtype;
MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
MPI_Send(a, 1, rowtype, dest, tag, MPI_COMM_WORLD);
```

Example 4

```
1  #define SIZE 4
2  float a[SIZE][SIZE] =
3    {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
4    9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
5  float b[SIZE];
6  MPI_Status stat;
7  MPI_Datatype rowtype;
8  MPI_Init(&argc,&argv);
9  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
11 MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
12 MPI_Type_commit(&rowtype);
13 if(numtasks == SIZE){
14   if(rank == 0)
15     for (i=0; i<numtasks; i++)
16       MPI_Send(&a[i][0], 1, rowtype, i, tag,
             MPI_COMM_WORLD);
17   /*the datatype rowtype can also be used in the
       following function*/
18   MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&
       stat);
19   //MPI_Recv(b,1,rowtype,source,tag,MPI_COMM_WORLD,&
       stat);
20   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
21         rank,b[0],b[1],b[2],b[3]);
22 }
23 MPI_Type_free(&rowtype);
24 MPI_Finalize();
```

Outline

Vector Type

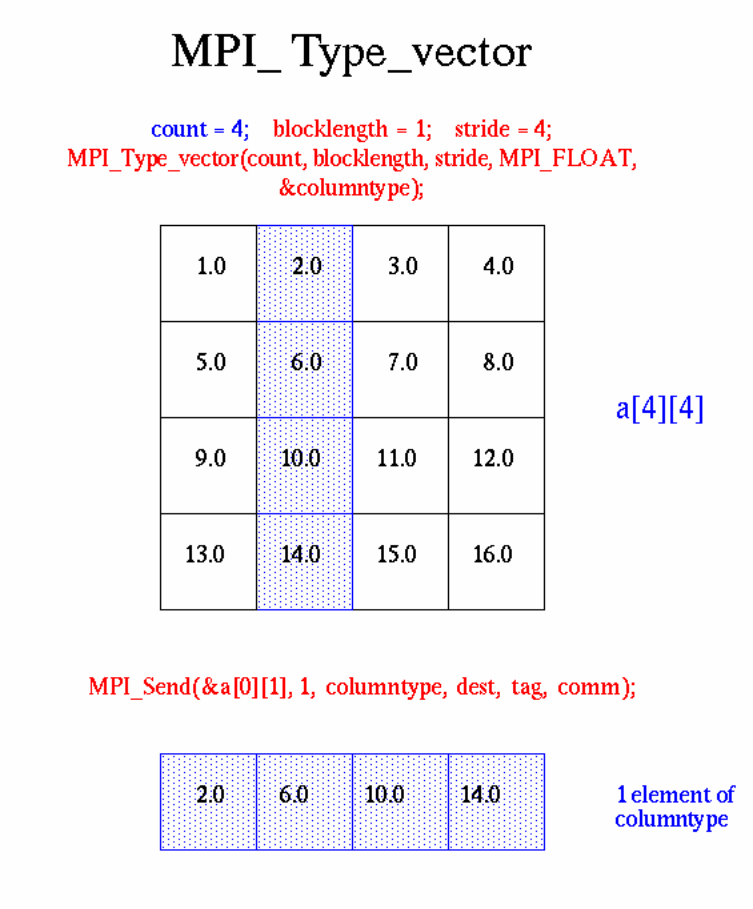
Vector: The vector datatype is similar to the contiguous datatype but allows for a constant non-unit stride between elements.

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

- Input parameters
  - `count`: number of blocks (nonnegative integer)
  - `blocklength`: number of elements in each block (integer)
  - `stride`: number of elements between each block (integer)
  - `oldtype`: old datatype

Vector Type Contd.

- Output parameter
  - `newtype`: new datatype



Vector Type Contd.

For example, the following two types can be used to communicate a single row and a single column of a matrix ($ny \times nx$):

```
MPI_Datatype rowType, colType;
MPI_Type_vector(nx, 1, 1, MPI_DOUBLE, &rowType);
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &colType);
MPI_Type_commit(&rowType);
MPI_Type_commit(&colType);
```

Example 5

```
1  #define SIZE 4
2  float a[SIZE][SIZE] =
3    {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
4    9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
5  float b[SIZE];
6  MPI_Status stat;
7  MPI_Datatype coltype;
8  MPI_Init(&argc,&argv);
9  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
11 MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &coltype);
12 MPI_Type_commit(&coltype);
13 if(numtasks == SIZE){
14   if(rank == 0){
15     for (i=0; i<numtasks; i++)
16       MPI_Send(&a[i][0], 1, coltype, i, tag,
             MPI_COMM_WORLD);
17   }
18   MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&
         stat);
19 }
20 MPI_Type_free(&coltype);
21 MPI_Finalize();
```

Outline

Indexed Type

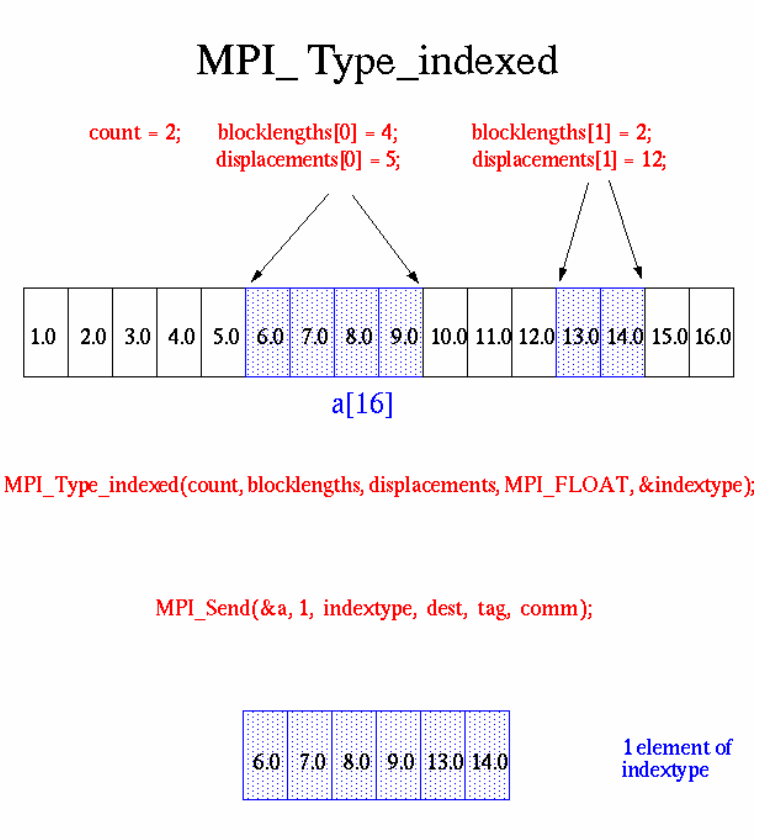
Indexed: The indexed datatype provides for varying strides between elements.

```
int MPI_Type_indexed(
                    int count,
                    int blocklens[],
                    int indices[],
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype )
```

- Input parameters
  - `count`: number of blocks — also number of entries in `indices` and `blocklens`
  - `blocklens`: number of elements in each block (array of nonnegative integers)
  - `indices`: displacement of each block in multiples of `oldtype` (array of integers)
  - `oldtype`: old datatype

Indexed Type Contd.

- Output parameters
  - `newtype`: new datatype



WITS UNIVERSITY

Indexed Type Contd.

Indexed type generalizes the vector type; instead of a constant stride, blocks can be of varying length and displacements.

```
int blocklen[] = {4, 2, 2, 6, 6};
int disp[] = {0, 8, 12, 16, 23};
MPI_Datatype mytype;
MPI_Type_indexed(5, blocklen, disp, MPI_DOUBLE,
                                        &mytype);
MPI_Type_commit(&mytype);
......
MPI_Type_free(&mytype);
```

Outline

Struct Type

Struct: The most general constructor allows for the creation of types representing general C/C++ structs/classes.

- We can use `MPI_Type_create_struct` to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
  int count, //number of elements in the datatype
  int array_of_blocklengths[], //length of each element
  MPI_Aint array_of_displacements[], //displacements in bytes
  MPI_Datatype array_of_types[],
  MPI_Datatype* new_type_p)
```

WITS UNIVERSITY

Struct Type cont.

- `count`: number of blocks, also number of entries in arrays `array_of_types`, `array_of_displacements` and `array_of_blocklengths`
- `array_of_blocklengths`: number of elements in each block
- `array_of_displacements`: byte displacement of each block
- `array_of_types`: type of elements in each block
- Output Parameters: `newtype`: new datatype

Struct Type Cont.

- To find the displacements, we can use the function `MPI_Get_address`:

```
int MPI_Get_address(
    void* location_p,
    MPI_Aint* address_p);
```

  - It returns the address of the memory location referenced by `location_p`.
  - `MPI_Aint` is an integer type that is big enough to store an address on the system.

Example 6 (Moving particles between processes)

In N-body problems, the force between particles become less with growing distance. At great enough distance, the influence of a particle on others is negligible. A number of algorithms for N-body simulation take advantage of this fact. These algorithms organize the particles in groups based on their locations using tree structures such quad-tree. One important step in the implementation of these algorithms is that of transferring particles from one process to another as they move. Here, we only discuss a way in which movement of particles can be done in MPI.

Assume a particle is defined by

```
typedef struct {
     int x,y,z;
     double mass;
}Particle;
```

- To send a particle from one process to another, or broadcast the particle, it makes sense in MPI to create a datatype instead of sending the elements in the struct individually.

Example 6 cont.

```
1    Particle my_particle;
2    MPI_Datatype particletype;
3    Build_mpi_type(&my_particle.x, &my_particle.y, &
         my_particle.z, &my_particle.mass, &particletype);
4    /*process 0 does some computation with my_particle */
5    ......
6    /*process 0 performs a broadcast*/
7    MPI_Bcast(&my_particle, 1, particletype, 0,
         MPI_COMM_WORLD);
8    ......
9    MPI_Type_free(&particletype);
10   }
```

Struct Type Contd.

Example 6 cont.

```
1  void Build_mpi_type( int* x_p, int* y_p, int* z_p, double
       * mass_p, MPI_Datatype* particletype_p) {
2    int array_of_blocklengths[4] = {1, 1, 1, 1};
3    MPI_Datatype array_of_types[4] = {MPI_INT, MPI_INT,
         MPI_INT, MPI_DOUBLE};
4    MPI_Aint array_of_displacements[4] = {0};
5    MPI_Get_address(x_p, &array_of_displacements[0]);
6    MPI_Get_address(y_p, &array_of_displacements[1]);
7    MPI_Get_address(z_p, &array_of_displacements[2]);
8    MPI_Get_address(mass_p, &array_of_displacements[3]);
9    for(int i=3; i<=0; i++)
10     array_of_displacements[i] -= array_of_displacements
           [0];
11   MPI_Type_create_struct(4, array_of_blocklengths,
12           array_of_displacements, array_of_types,
13           particletype_p);
14    MPI_Type_commit(particletype_p);
15 }  /* Build_mpi_type */
```

Summary

-

WITS UNIVERSITY

References

Useful references:

- Trobec et al., *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*
- Quinn, *Parallel Programming in C with MPI and OpenMP*
- Grama et al., *Introduction to Parallel Computing*
- Barney, *Message Passing Interface*
- MPI, *MPI Forum*

References I

- Barney, Blaise. *Message Passing Interface*. https://hpc-tutorials.llnl.gov/mpi/. Accessed 2021-10-6.
- Grama, Ananth et al. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- MPI. *MPI Forum*. https://www.mpi-forum.org/docs/. Accessed 2021-10-6.
- Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.
- Trobec, Roman et al. *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms*. Springer, 2018.