**Data distribution (regular vs. irregular)**

**Regular** distribution of arrays:

```c
int NGA_Create(int type, int ndim, int dims[], char *array_name, int chunk[])
```

- `array_name` - a unique character string [input]
- `type` - data type (MT_F_DBL, MT_F_INT, MT_F_DCPL) [input]
- `ndim` - number of array dimensions [input]
- `dims[ndim]` - array of dimensions [input]
- `chunk[ndim]` - array of chunks, each element specifies minimum size that given dimensions should be chunked up into [input]

**chunk** indicates the distribution size for all or some of array dimensions. E.g., for a 2-dimensional array, 1) setting `chunk[0]=dim[0]` gives distribution by vertical strips; 2) setting `chunk[1]=dim[1]` gives distribution by horizontal strips.

Actual chunks will be modified so that they are at least the size of the minimum and each process has either zero or one chunk. Specifying `chunk[i]` as <1 will cause that dimension to be distributed evenly.

**Irregular** distribution of arrays:

```c
int NGA_Create_irreg(int type, int ndim, int dims[], char *array_name, int block[], int map[])
```

- `array_name` - a unique character string [input]
- `type` - MA data type (MT_F_DBL, MT_F_INT, MT_F_DCPL) [input]
- `ndim` - number of array dimensions [input]
- `dims` - array of dimension values [input]
- `block[ndim]` - no. of blocks each dimension is divided into [input]
- `map[s]` - starting index for each block; the size `s` is a sum all elements of nblock array [input]

Creates an array by following the user-specified distribution and returns integer handle representing the array.

The distribution is specified as a Cartesian product of distributions for each dimension. The array indices start at 0. For example, the following figure demonstrates distribution of a 2-dimensional array 8x10 on 6 (or more) processors. `block[2]={3,2}`, the size of `map` array is `s=5` and array `map` contains the following elements `map={0,2,6, 0, 5}`. The distribution is nonuniform because, P1 and P4 get 20 elements each and processors P0,P2,P3, and P5 only 10 elements each.
Ghost arrays

Arrays have data which are padded by a boundary region of array elements representing portions of the global array residing on other processors. These boundary regions can be updated with data from neighboring processors by a call to a single GA function. In order to create such global arrays, there are two calls for regular and irregular distribution respectively:

1) \texttt{int NGA\_Create\_ghosts(int type, int ndim, int dims[], int width[], char *array\_name, int chunk[])}

2) \texttt{int NGA\_Create\_ghosts\_irreg(int type, int ndim, int dims[], int width[], char *array\_name, int map[], int block[])}

\texttt{width} is used to control the width of the ghost cell boundaries in each dimension of the global array. Different dimensions can be padded with different numbers of ghost cells, although it is expected that for most applications the widths will be the same for all dimensions.

For a global array with ghost cells, the data distribution can be visualized as follows:

“ghost cell” can be updated in a single call to \texttt{nga\_update}. To access the data in the ghost cells, the user must use the \texttt{nga\_access\_ghosts} function.

\begin{center}
\texttt{Global Array with Ghost Cells}
\end{center}
Non-blocking one-sided communication

The non-blocking operations (get/put/accumulate) are derived from the blocking interface by adding a handle argument that identifies an instance of the non-blocking request. The operation can be completed locally by making a call to the wait (e.g. nga_nbwait) routine.

**Example:** Let us take a simple case for illustration. Say, there are two global arrays i.e. one array stores pressure and the other stores temperature. If there are two computation phases (first phase computes pressure and second phase computes temperature), then we can overlap communication with computation, thus hiding latency.

```
nga_get (get_pressure_array)

nga_nbget(initiates data transfer to get temperature_array, and returns immediately)

compute_pressure() /* hiding latency - communication is overlapped with computation */

nga_nbwait(temperature_array - completes data transfer)

compute_temperature()
```

```c
void NGA_NbPut(int g_a, int lo[], int hi[], void *buf, int ld[],
                 ga_nbhdl_t* nbhandle)

void NGA_NbGet(int g_a, int lo[], int hi[], void *buf, int ld[],
                ga_nbhdl_t* nbhandle)

void NGA_NbAcc(int g_a, int lo[], int hi[], void *buf, int ld[],
                void *alpha, ga_nbhdl_t* nbhandle)

int NGA_NbWait(ga_nbhdl_t* nbhandle)
```