YAXT as a coupler (Yet Another eXchange Tool, DKRZ 2023)

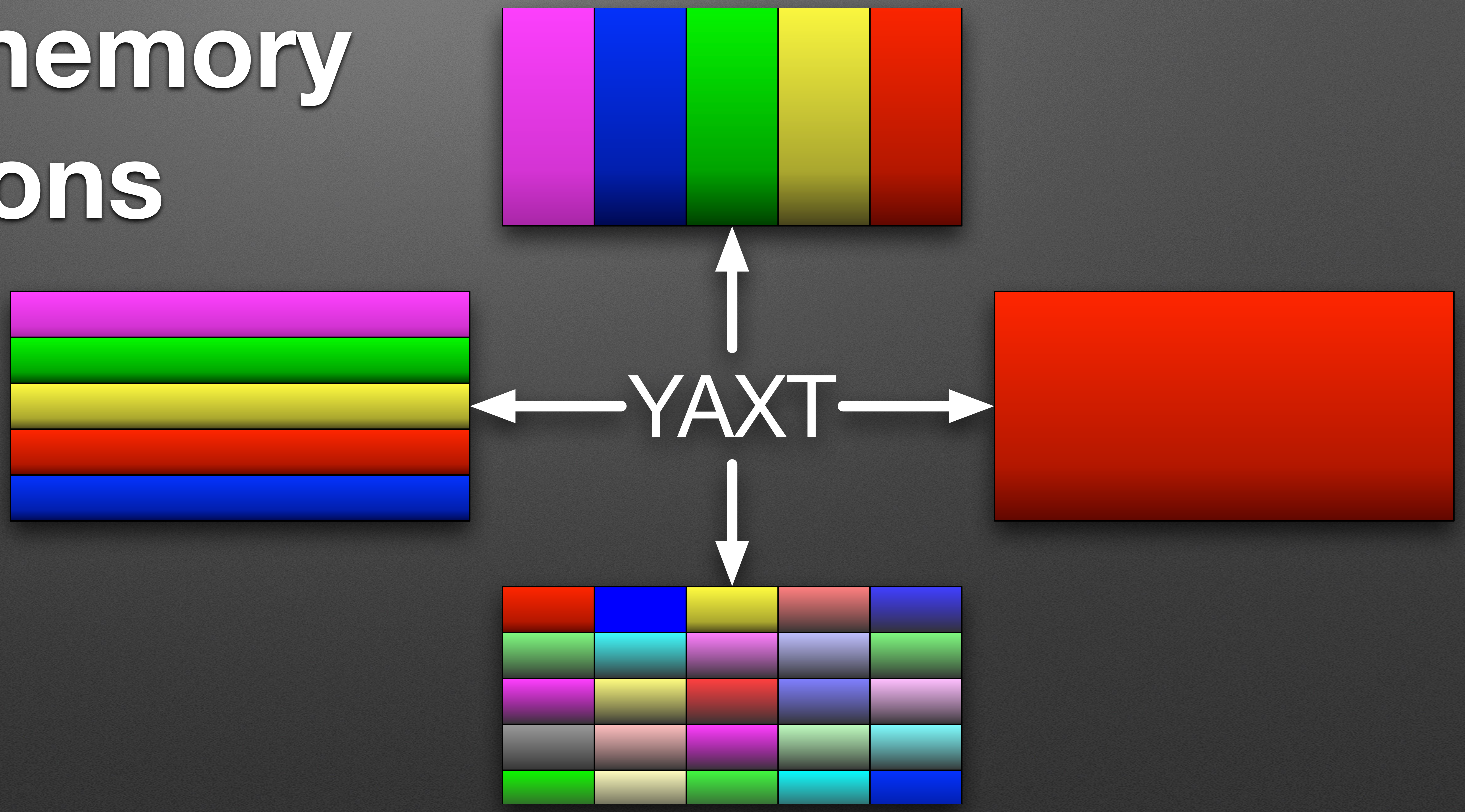
Thomas Jahns, DKRZ

Credits to Moritz Hanke and Jörg Behrens

Outline

- Design Overview
- User perspective, focus on concepts
- More detailed description of individual parts and basic examples
- Applying this to an ESM, ICON example

Automated data redistributions in distributed memory applications



Where this comes from

- Project ScalES in 2009-2012 had shown how programmable communications simplified several tasks in model development
- Reusable components prevent re-inventing the wheel for every model we touch

Design targets and constraints

- Ease programming of all redistributions of data in MPI programs
    - Attempt to capture all climate model use cases
- Fully scalable
- Easy to use correctly
- Adaptable to any data decomposition
- Convenient Fortran interface
- Type agnostic

Implementation choices

- Library, core implemented in C
- Fortran interface based on F2003
- Uses libtool and pkg-config to ease linking to
- BSD 3-clause license
- Rely on MPI datatypes to describe elements
- Describe set of element with global integer IDs (compile time choice of width)

Resources

- DKRZ Gitlab: https://gitlab.dkrz.de/dkrz-sw/yaxt/
- Redmine: https://swprojects.dkrz.de/redmine/projects/yaxt/
- Doxygen documentation: https://dkrz-sw.gitlab-pages.dkrz.de/yaxt/

Setup phases

- On each participating MPI rank:

  - Create objects describing local lists of present and present-to-be elements
  - Compute necessary communication in type-independent fashion
  - Derive type-specific communication object
  - Perform data exchange

  - xt_idxvec_new, xt_idxstripes_new, xt_idxsection_new, …
  - xt_xmap_dist_dir_new, xt_xmap_all2all_new
  - xt_redist_p2p_new, xt_redist_collection_new
  - xt_redist_s_exchange, xt_redist_a_exchange

Preventing leaks

For each class, there is a corresponding delete call:

- `CALL xt_idxlist_delete(idxlist)`
- `CALL xt_xmap_delete(xmap)`
- `CALL xt_redist_delete(redist)`

Defining a decomposition
Prerequisites

- In the usual and simple case, data elements have the same memory layout and are stored in a single C object (think sequence association).
- Each data element can be referred to by a unique global id (integer).
- The local part of a decomposition is a list of global data element ids.
- The positions of the global ids within the set correspond to the positions of the respective data elements within the data array (if nothing else is specified).

2D-example: Specifying the source distribution

```
src_idxlist =

xt_idxvec_new((/ 1, 2, 5, 6, 9, 10 /))

xt_idxsection_new(1, 2, (/ 6, 4 /), (/ 3, 2 /), &
                  (/ 0, 0 /))

xt_idxfsection_new(1, 2, (/ 4, 6 /), (/ 2, 3 /), &
                   (/ 1, 1 /))

xt_idxstripes_new((/ xt_stripe(1,2,1), &
                     xt_stripe(5,2,1), &
                     xt_stripe(9,2,1) /))

xt_idxstripes_new((/ xt_stripe(1,3,4), &
                     xt_stripe(2,3,4) /)
```

Don't do that! Without deliberation, that is.

2D-example: Specifying the destination distribution
Halo-exchange

```
dst_idxlist =

xt_idxvec_new((/ 3, 7, 11, 13, 14, 15 /))

xt_idxstripes_new((/ xt_stripe(3,3,4), &
                     xt_stripe(13,3,1) /))
```

2D-example: Specifying the destination distribution
Gather

Memory costs are someone else's problem 😉

dst_idxlist =

```
xt_idxvec_new((/ 1, 2, …, 23, 24 /))

xt_idxsection_new(1, 2, (/ 6, 4 /), (/ 6, 4 /), &
                  (/ 0, 0 /))

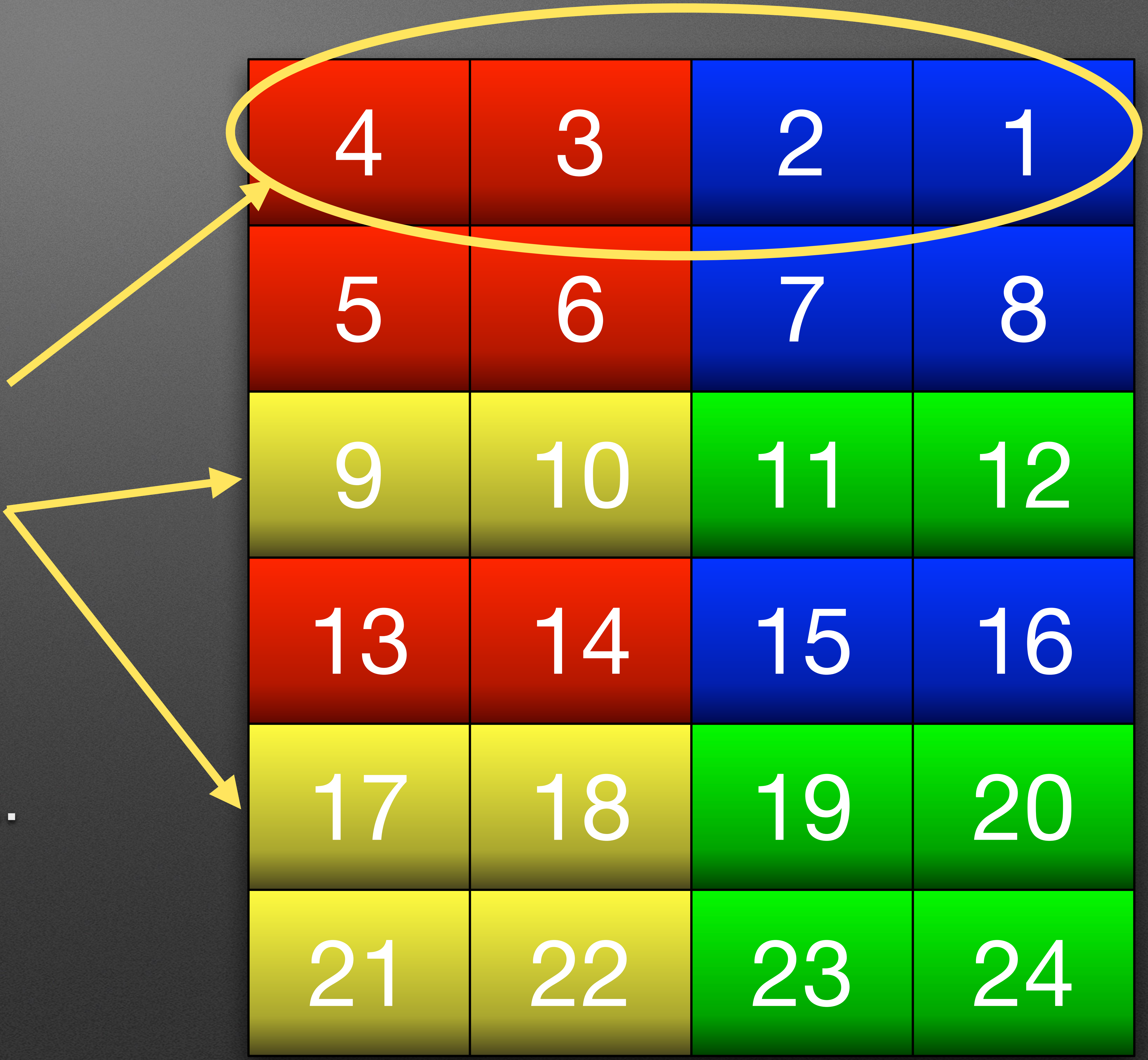
xt_idxfsection_new(1, 2, (/ 4, 6 /), (/ 4, 6 /), &
                   (/ 1, 1 /))

xt_idxstripes_new((/ xt_stripe(1,24,1) /))
```

Every other rank: `xt_idxempty_new()`

The domain decomposition transcends the ordinary

- xt_idxmod allows creation of derived index lists, where some indices are substituted
- xt_idxlist_collection enables concatenation of index lists
- You can think of it, YAXT has you covered.
  Pinky swear.
  Round-robin, dynamically load-balanced, …



Index list recap

- Convenient description of finite sequence of integer values
- Value semantics
- Operations for different representations and intersections*
- Strictly rank-local, non-collective methods

Generate a mapping between source and target decomposition

One call to compute all transfers necessary:

```
xmap = xt_xmap_dist_dir_new(src_idxlist, &
                            dst_idxlist, comm)
```

For the sake of completeness:

- `xt_xmap_dist_dir_new` — Uses the Rendezvous algorithm to be scalable in both time and space
- `xt_xmap_all2all_new` — If the intersections are both dense and cheap to compute, or for a better debugging experience because of the more naive algorithm
- `xt_xmap_intersection_new` — If an alternative means to arrive at intersections is available

Exercises for the advanced reader:

- `xt_xmap_dist_dir_new` and `xt_xmap_all2all_new`:
  - comm can be an intercomm
- More interesting than gather:
  - Partial gather
  - Scatter-Gather
  - Combine Gather and Transposition
- Improve efficiency by marking the communicator to be not used in conflicting fashion (`xt_mpi_comm_mark_exclusive(comm)`)

Xmap recap

- Contains abstract description of communication matrix
- Most operations collective for all MPI ranks involved in creation
- Constructor solves all the hard problems of creating the communication matrix

Specific redistribution object: Xt_redist

- Redists can be built for every non-null MPI data type (basic types, structs, vectors, ...)
- Internally YAXT will build MPI data type for every required exchange
  ➡ no buffers are required for the exchange on caller side
- For a combined redistribution object YAXT can also build MPI data types even if the associated input arrays have no fixed offset between each other

From xmap to redist:

- Concretize xmap for single data type:
  - ```
    redist_a = xt_redist_p2p_new(xmap, &
                                 MPI_REAL)
    ```
  - ```
    redist_b = xt_redist_p2p_new(xmap, &
                                 MPI_INTEGER)
    ```

Handling memory layout

- `xt_redist_p2p_off_new` — Specify offsets per element of index lists used to form the xmap (offsets are to be interpreted in terms of element size)
- `xt_redist_p2p_blocks_new` — Each element is a contiguous block of variable size
- `xt_redist_p2p_blocks_off_new` — Combination of the above
- `xt_redist_p2p_ext_new` — Specify offsets as extents, i.e. `c_int` start, size and stride (or `MPI_AINT` start and stride for `xt_redist_p2p_aext_new`)

Adapting to system properties

```
USE YAXT
TYPE(xt_xmap) :: xmap
TYPE(xt_config) :: conf
TYPE(xt_redist) :: redist_c
conf = xt_config_new()
! OpenACC GPU kernel handles datatype packing
CALL xt_config_set_exchange_method(conf, xt_exchanger_irecv_isend_ddt_packed)
! prerequisite MPI_Init_thread(MPI_THREAD_MULTIPLE)
! call MPI_Send, MPI_Recv etc. in OMP PARALLEL DO
CALL xt_config_set_redist_mthread_mode(conf, XT_MT_OPENMP)
redist_c = xt_redist_p2p_custom_new(xmap, MPI_DOUBLE_PRECISION, conf)
```

Redists for aggregation:

- Build single transfer for multiple arrays:
  - ```
    redist_c = xt_redist_collection_new((/ redist_a, redist_b), &
                                          cache_size, comm)
    ```
    (arrays where relative memory positions are flexible)
  - ```
    redist_d = &
      xt_redist_collection_static_new((/ redist_a, redist_b /), &
                                       src_displacements, dst_displacements, comm)
    ```
    (array's relative memory positions always the same)
- Apply the same redist to multiple (sub-)arrays:
  - ```
    redist_e = xt_redist_repeat_new(redist_a, src_extent, dst_extent, &
                                     displacements)
    ```
  - displacements different for source and destination: `xt_redist_repeat_asym_new`
    ```

Moving data

A. Synchronous:

   `xt_redist_s_exchange`

B. Asynchronous:

   `xt_redist_a_exchange`

   `xt_request_wait`

Actual redist calls:

- `CALL xt_redist_s_exchange(redist, C_LOC(src), C_LOC(dst))`
- ```
  CALL xt_redist_s_exchange(redist, (/ C_LOC(src1), C_LOC(src2) /), &
                                    (/ C_LOC(dst1), C_LOC(dst2) /) )
  ```
- If the type of src and dst is one of INTEGER(i4), INTEGER(i8), REAL(dp), REAL(sp), or LOGICAL:
  `CALL xt_redist_s_exchange(redist, src, dst)`
- ```
  CALL xt_redist_a_exchange(redist, C_LOC(src), C_LOC(dst), request)
  ! intermediate computation not touching src or dst here
  CALL xt_request_wait(request)
  ```

Redist recap

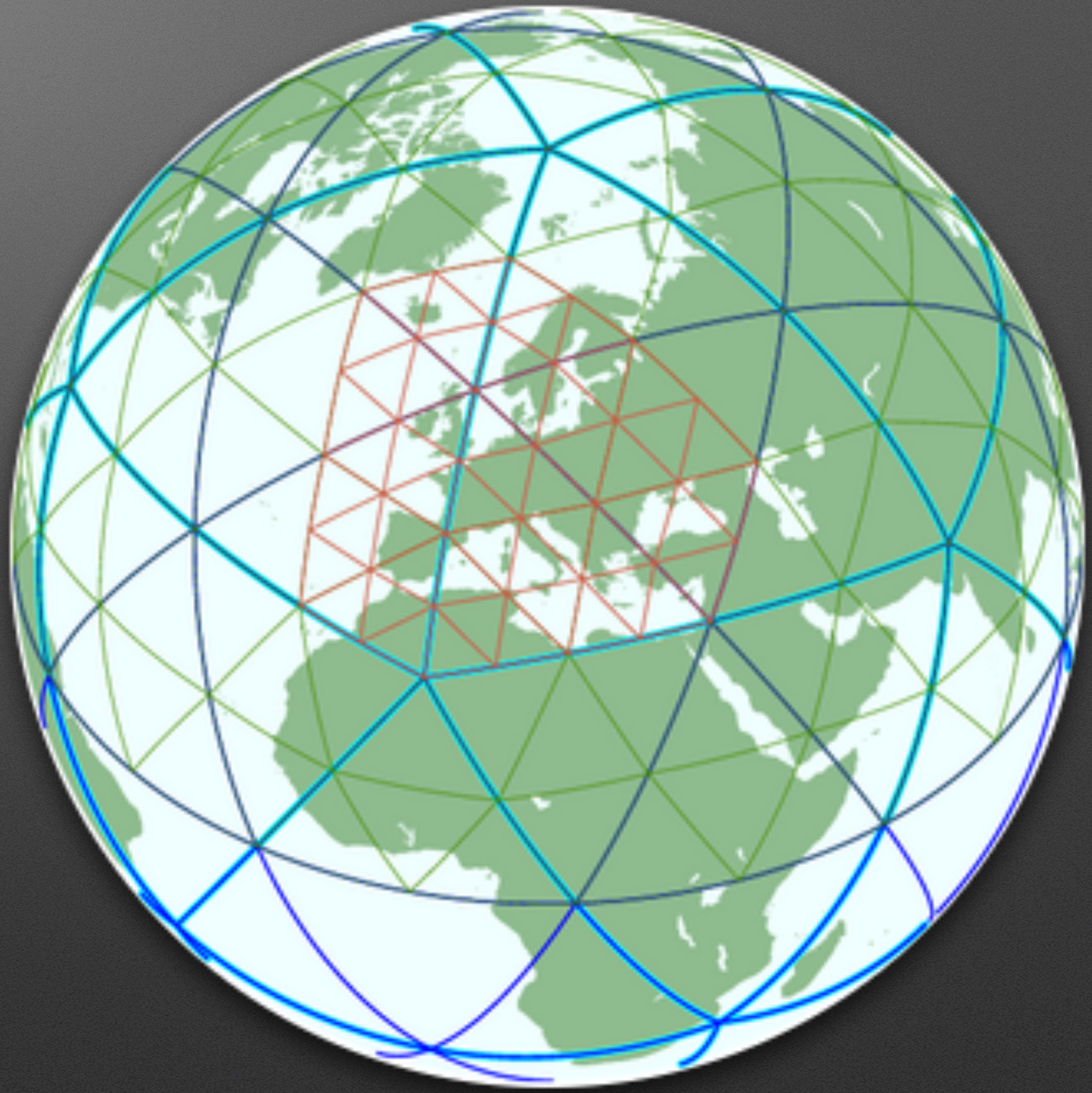
- Implements specific transfers for data sets according to communication matrix and involved concrete data types
- Contains full message scheduling logic, buffering, progress
- Ideally, construction overhead can be recaptured by repeated use
- Encapsulates internal exchanger object
- Support of GPUs

ICON decomposition

- t_patch contains everything needed for an index vector:

```
%n_patch_cells
%cells%decomp_info%glb_index,
%cells%decomp_info%glb2loc_index,
%cells%decomp_info%owner_local
```

- Substitute `verts` or `edges` for `cells` when needed.



Key concepts rehash

- Use index lists describing array contents of present (source) and future (target) decomposition
- Create xmap to derive needed communication partners and message contents in terms of abstract elements
- Concretize redists from xmap and MPI datatypes for individual arrays, build redist collections for message aggregation

YAXT is flexible

- Some constructors come in a version that takes a config object (`xt_config_new`) to override defaults (set by environment variables):
  - Pick exchanger (`XT_CONFIG_DEFAULT_EXCHANGE_METHOD`)
  - Activate multi-threading (`XT_CONFIG_DEFAULT_MULTI_THREAD_MODE`)
  - Stop automatic index vector conversion (`XT_CONFIG_DEFAULT_IDXVEC_AUTOCONVERT_SIZE`)
  - More to come…

YAXT is modular and open to extension

Nearly all parts of YAXT can be easily substituted:

- Have some better method to derive the communication matrix?
  Use xt_xmap_intersection_new

- Already have MPI datatypes for all messages but want to aggregate communication? Use
  xt_redist_single_array_base+xt_redist_collection(_static)

- Know a better way to schedule messages? Write your own exchanger

Thank you for your attention.

Questions?