Halo Exchange Example

This Jupyter notebook presents an example about performing halo exchanges in diezDecomp.

Since halo exchanges are relatively simple operations, this example corresponds to a general case, where multiple variables are randomized:

- Halo exchange direction (`ii`) in the global $x/y/z$ coordinate system ($x \to 0,\ y \to 1,\ z \to 2$).
- Number of halo cells in every direction (`nh_xyz`).
- Array padding for each MPI task (`offset6`).
- 3D array size (`n3`).
- Periodicity in the halo exchange direction (`periodicity_xyz`)
- Custom order for the local data (`order_halo`) (e.g., `x-y-z` or `y-z-x`).
- Grid layout for the pencil distribution of the MPI tasks.

To perform an exhaustive study, a Python script (`gen_random_halos.py`) generates random combinations of the previous variables, and the Fortran code is called. The results are verified by pre-computing the expected values after the halo exchange (`A_ref`), and checking with the results after the operation.

In this example, most of the Fortran code is about defining the analytical values expected for `A_ref` . Only 3 lines of code are required for calling diezDecomp:

```
call diezdecomp_track_mpi_decomp(lo_ref, obj_ranks, irank, nproc)
call diezdecomp_generic_fill_hl_obj(hl, obj_ranks, A_shape, offset6, ii, nh_xyz, order_halo,
periodic_xyz, wsize, use_halo_sync, autotuned_pack)
call diezdecomp_halos_execute_generic(hl, A, work)
```

The first subroutine (`diezdecomp_track_mpi_decomp`) is used to define the grid layout for the MPI tasks, which is stored in the object `obj_ranks` . Then, the descriptor for the halo exchange (`hl`) is initialized using the subroutine `diezdecomp_generic_fill_hl_obj` . Finally, during the CFD iterations, the subroutine `diezdecomp_halos_execute_generic` performs the halo exchange using the object descriptor `hl` , the input array `A` , and a small work buffer `work` .

Summary API Input

A summary about all input variables for the diezDecomp subroutines in this example is given below:

- `nproc` is the total number of MPI processes.
- `irank` is the global rank of each MPI task.
- `lo_ref(0:2)` is a reference `[i,j,k]` position for each MPI task in the global 1D/2D/3D pencil distribution for the simulation domain.
  - Based on the `lo_ref` coordinates, diezDecomp automatically tracks the location of each MPI task in the physical domain.
  - Any reference coordinate can be used as `lo_ref(0:2)` , as long as it properly describes the location of each MPI task in the 1D/2D/3D pencil distribution.
    - However, please note that all MPI tasks aligned along a Cartesian direction ($x/y/z$) are required to have the same coordinate (e.g. `lo_ref(0)=i=0` for the first slice along the x-direction).
- `ii` is the global direction of the halo exchange operation: ($x \to 0,\ y \to 1,\ z \to 2$).
- `nh_xyz` is the number of halo cells along the $x/y/z$ directions.
  - While the current halo exchange operation is along the `ii` -direction, DNS/LES solvers will typically have arrays with reserved halo cells along multiple directions at once.
- `periodic_xyz` is the presence of periodic boundary conditions along the $x/y/z$ directions.
  - Please note that only the periodicity along the `ii` -direction will be considered for the halo exchange operation.
  - Other periodic boundary conditions might be used for subsequent halo exchanges.
- `order_halo` is the local order for the input array `A` .
  - For instance, `order_halo=[1,2,0]` implies that the array `A` has an indexing system ordered in the $y/z/x$ directions.
- `offset6` is the (ignored) padding for the input array `A` in each dimension:
  - `offset6(i,:,0)` is the padding at the beggining of the dimension `i` , whereas `offset6_in(i,:,1)` is the padding added at the end of such dimension.
  - This padding is external to the reserved cells for the halo exchange (`nh_xyz`):
    - The cells described by `offset6` are ignored by the halo exchange system.
    - DNS/LES solvers can require extra padded cells (`offset6`) for various practical reasons, such as re-using an allocated array for another task.
  - Please note that `offset6` follows the order specified by `order_halo` .
- `A_shape` is the shape of the input array `A` , including the (ignored) padded cells `offset6` and the reserved halo cells `nh_xyz` .
  - The order of `A_shape` is consistent with `order_halo` .
- `use_halo_sync` is a Boolean flag indicating if synchronous halo exchange operations are needed.
  - Most applications work faster with `use_halo_sync = .false.` , but the option is still available. The performance differences are minor.
- `autotuned_pack` is a Boolean flag to auto-tune the data packing algorithm for the halo exchange.
  - Usually, data packing operations in halo exchanges are very fast compared to MPI transfers, and thus this auto-tuning feature can be disabled.

Additional notes:

- The variable `wsize in diezdecomp_generic_fill_hl_obj` is a secondary output, indicating the minimum size of the work buffer (`work`) needed by the halo exchange operation.
  - The user is responsible for ensuring that the size of the `work` array is larger than `wsize` .
  - If multiple halo exchanges are performed, `diezdecomp_generic_fill_hl_obj` can be called during initilization (without allocated arrays) to identify the maximum value of `wsize` .
- In the Fortran input files, the variable `flat_mpi_ranks` corresponds to a special array, where `flat_mpi_ranks(rank,:)` indicates the global `[i,j,k]` position of the MPI task `rank` within the global pencil distribution.
  - If such array is available, it is one of the best choices to define `lo_ref(0:2)` for the subroutine `diezdecomp_generic_fill_hl_obj` .

Autotuning for Halo Exchanges

In diezDecomp, two autotuning options are available for halo exchanges:

- MPI data transfer mode (synchronous vs. asynchronous).
- Data packing/unpacking algorithm (batched vs. simultaneous).

Each option is described separately below:

MPI data transfer mode

In diezDecomp, halo exchanges can be performed using either asynchronous (`MPI_ISend/IRecv`) MPI operations for data transfer, or synchronous (`MPI_SendRecv`) pairs. Usually, asynchronous transfers are slightly faster than synchronous operations, yet both options are available for autotuning.

The `integer` parameters to control MPI transfers are:

- CaNS API: global variable `diezdecomp_halo_mpi_mode` .
- Generic API: input variable `force_halo_sync` for the subroutine `diezdecomp_generic_fill_hl_obj` .

For both variables (`diezdecomp_halo_mpi_mode and force_halo_sync`), the following conventions are used:

- `1` : Synchronous MPI operations (`MPI_SendRecv`).
- `2` : Asynchronous MPI transfers (`MPI_ISend/IRecv`).
- Other values: enable autotuning.

By default, the CaNS API is configured to use asynchronous transfers (`MPI_ISend/IRecv`). Autotuning must be enabled manually by changing `diezdecomp_halo_mpi_mode` .

Data packing/unpacking algorithm

In halo exchanges, the main performance bottleneck are usually MPI data transfers. However, performance improvements can also be found by optimizing other operations. For example, MPI transfers require 1D data buffers, containing all information sent or received by the operation. Packing and unpacking data from these buffers requires especialized GPU kernels.

In diezDecomp, two options are available for data packing/unpacking from GPU kernels.

- "Batched" mode:
  - Separate GPU kernels are launched to pack/unpack data from every slice in the input array (`A(i,j,k)`) participating in the halo exchange.
- "Simultaneous" mode:
  - Only one GPU kernel is launched to pack/unpack data from the entire MPI 1D buffer.

Both of the previous options are available for autotuning ("batched" vs. "simultaneous"). Generally speaking, the "batched" mode has GPU kernels with shorter instructions, and it is (slightly) faster for halo exchanges requiring few cells (e.g. `nh_xyz = [2,2,2]`). The "simultaneous" mode has GPU kernels with generalized code, and it tends to be faster for halo exchanges requiring many cells (e.g. `nh_xyz >> [2,2,2]`).

The variables for controlling the packing/unpacking behavior of the halo exchange are[1]:

- CaNS API: global variable `diezdecomp_halo_autotuned_pack` .
- Generic API: input variable `autotuned_pack` in the subroutine `diezdecomp_generic_fill_hl_obj` .

The conventions for the variables `diezdecomp_halo_autotuned_pack and autotuned_pack` are:

- `2` : "batched" mode.
- `3` : "simultaneous" mode.
- Other values: enable autotuning.

By default, the CaNS API enables the "batched" mode (`2`), because it tends to be faster for the halo exchanges found in the CaNS project. However, please note that the "simultaneous" mode (`3`) could be faster for DNS/LES solvers using high-order methods requiring large halo exchanges.

[1]: The numbers `2 and 3` are a reference to the "batched" mode (`2`) working with 2D slices, and the "simultaneous" mode (`3`) processing entire 3D arrays.

Autotuning Report (Halo Exchanges)

In diezDecomp, all benchmark results for autotuning operations (with halo exchanges) are recorded inside the object (`hl`). A summary of the results can be printed with the subroutine `diezdecomp_summary_halo_autotuning` , without performing any additional benchmarks. This allows the user to better understand how different autotuning choices influence the speed of halo exchanges, and develop practical guidelines for manually choosing parameters.

Due to the limited number of autotuning parameters for halo exchanges, please note that most DNS/LES solvers can operate with fixed choices delivering high performance:

- Synchronous vs asynchronous MPI transfers.
- Batched vs. simultaneous data packing/unpacking modes.

Further Information

After analyzing the previous examples, further details about diezDecomp halo exchanges can be found in the specialized test suite: `./tests/halo` .