Transpose Examples

This Jupyter notebook presents three examples performing transposes in diezDecomp:

1. A simple `x-y` transpose with a grid layout (for the MPI tasks) going from $(p_x \times p_y \times p_z) = (1 \times 2 \times 4)$ to $(2 \times 1 \times 4)$ (in the x y z directions).
   - This is a basic use-case for DNS/LES simulations.
2. Advanced example with randomized grid layout for 8 GPUs/GCDs. The following challenges are addressed:
   - Random transpose direction (`x/y/z`)
   - Mismatched intersections between the 2D pencil decompositions.
   - Custom array padding for each MPI task.
   - Array order (e.g., `x-y-z` or `y-z-x`) is different for the input/output arrays.
3. Optional example, performing the same non-trivial transposes from Example 2, but using an advanced subroutine which does not require an additional memory buffer (`work`).
   - The restrictions for using this subroutine are explained in detail below (Example 3).
   - While not all applications can operate without a memory buffer (`work`), Poisson solvers and parallel tridiagonal solvers can often meet the required conditions, without interfering with the rest of the DNS/LES solver.

The generic API is considered for all transpose examples (`diezdecomp_api_generic.f90`).

Example 1: Simple X-Y Transpose

This example corresponds to one of the most simple use-cases encountered in DNS/LES solvers:

- Performing a X-Y transpose between well-aligned 2-D pencil distributions.

Globally, the grid size is:

- $(N_x \times N_y \times N_z) = (200 \times 320 \times 440)$

The original (x-aligned) grid layout for the MPI tasks is:

- $(p_x \times p_y \times p_z) = (1 \times 2 \times 4)$

After the transpose, the MPI grid layout is (y-aligned):

- $(p_x \times p_y \times p_z) = (2 \times 1 \times 4)$

The local grid size of each MPI task is:

- $size(p_{in}) = 200 \times 160 \times 110$
- $size(p_{out}) = 100 \times 320 \times 110$

Please note that p_z is constant for both cases, and thus there is no data transfer along the z-direction.

Typically, DNS codes perform x-y transposes inside Poisson solvers. First, a local 1D FFT operation is applied in the x-direction, then a x-y transpose is called, and then a FFT is performed in the y-direction. Since 1D FFTs work faster when the entire 1D array is local to each GPU, it is common for DNS solvers to require that x-y transposes use $p_x = 1$ for the input array (`p_in`), and $p_y = 1$ in the output array (`p_out`). This ensures that information is re-distributed, such that each GPU thread can read all the required data for the 1D FFT operations in the x and y directions.

The correctness of the x-y transpose is verified by pre-computing the expected values for each element in the arrays `p_in` and `p_out_ref` inside the code. (The array `p_out_ref` is never part of the transpose operation). Finally, the transpose operation is validated by checking the maximum difference between `p_out` and `p_out_ref` .

Example 2: Generalized Transposes

In this example, it is shown how diezDecomp can handle arbitrary transpose operations with complex conditions, such as:

- Mismatched intersections between the 2D pencil decompositions.
- Custom array padding for each MPI task.
- Array order (e.g., `x-y-z` or `y-z-x`) that is different for the input/output arrays.
- Random transpose directions (e.g. $x \rightarrow y$, $z \rightarrow x$, etc.)

To simplify the process, the python script `gen_random_transposes.py` generates random configurations, which are read by Fortran (as an input file) and are applied. The results are verified as before, by pre-computing the expected values for `p_in` and `p_out_ref` .

This example can be considered as a baseline to use diezDecomp transposes in non-trivial settings.

Notes about the input:

- `order_in` is the local order (e.g. `y-z-x`) for the data in the input array `p_in` .
  - Similarly, `order_out` is the local order for the output array `p_out` .
  - `order_intermediate` is the local order used for packing/unpacking inside diezDecomp.
    - The final results never depend on `order_intermediate` , but some settings offer higher performance.
- `flat_mpi_ranks_in` is an array that specifies the location of each MPI task in the global grid layout (for the input array `p_in`):
  - For example, `flat_mpi_ranks_in(2,:) = [3,4,5]` means the `MPI_task=2` uses the position `[3,4,5]` in the grid layout for the pencil distribution: $(p_x \times p_y \times p_z)$.
  - The values of `flat_mpi_ranks_in` are always given in the global $x/y/z$ coordinates. The local orders (`order_in/order_out`) are not considered, since this is a global descriptor.
  - `flat_mpi_ranks_out` follows the same logic (as `flat_mpi_ranks_in`), but it specifies the location of each MPI task for the output array `p_out` .
  - Please note that diezDecomp can handle any arbitrary values for `flat_mpi_ranks_*` (even randomized).
- `offset6_in` is the padding for the input array `p_in` in each dimension:
  - `offset6_in(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.
  - `offset6_out` follows the same conventions, but for the output array `p_out` .
  - Please note that both `offset6_in` and `offset6_out` follow the orders given by `order_in` and `order_out` respectively.
- `n3_in` is the local array size for `p_in` , but following the local order given by `order_in` .
  - In this example, `n3_in` is the inner array size excluding `offset6_in` , so both quantities are added inside the Fortran code (when needed).
    - However, please note that the array shapes passed to diezDecomp (`sp_in` and `sp_out`) include the padding arrays (`offset6_in` and `offset6_out`).
  - Similarly, `n3_out` specifies the grid size for `p_out` (excluding `offset6_out`).

API Summary (Example 2)

In this example, it is shown how diezDecomp can be applied to nontrivial settings, by calling only 4 commands:

```
! recover grid layout for MPI tasks, starting from lo_in/out
call diezdecomp_track_mpi_decomp(lo_in , obj_rank_in , irank, nproc, order_in)
call diezdecomp_track_mpi_decomp(lo_out, obj_rank_out, irank, nproc, order_out)

! initialize the transpose descriptor (tr)
call diezdecomp_generic_fill_tr_obj(tr, obj_rank_in, obj_rank_out, sp_in, offset6_in, sp_out,
offset6_out, order_in, order_out, order_intermediate, allow_alltoallv, i_a2av, j_a2av, wsize,
allow_autotune_reorder)

! call diezDecomp (only this command is needed inside DNS/LES iterations)
call diezdecomp_transp_execute_generic_buf(tr, p_in, p_out, work)
```

The inputs for the code are once again summarized below:

- `lo_in`
- `lo_out`
- `irank` (global rank of MPI task)
- `nproc` (global number of MPI processes)
- `order_in` (explained above)
- `order_out`
- `order_intermediate`
- `sp_in` (3D shape of `p_in` , including padding)
- `sp_out` (3D shape of `p_out` , including padding)
- `offset6_in`
- `offset6_out`
- `allow_autotune_reorder` (autotuning option, can be set to false) (please see notes below about autotuning)
- `allow_alltoallv` (classic MPI alternative, can be set to false)
- `i_a2av` (only read if allow_alltoallv is active)
- `j_a2av` (only read if allow_alltoallv is active)
- Note:
  - `wsize` is a secondary output, with the size of the work buffer for `diezdecomp_transp_execute_generic_buf` .

In Example 3, it will be shown that diezDecomp includes a transpose mode which does not require an independent memory buffer (`work`), but it is subject to significant restrictions that the user is responsible for checking.

Example 3 (Advanced): Generalized Transposes without Work Buffer

(Optional Example, Subject to Strong Restrictions)

In the previous example (2), diezDecomp is called using a `work` array buffer, with enough space to store both the input and output arrays (`total_size(p_in)+total_size(p_out)`).

This is necessary to ensure that:

- There is always enough space to store auxiliary variables.
- All data in the arrays `p_in` and `p_out` is fully preserved, and the padded cells positions `offset6_in/out` are never modified:
  - The data in `p_in` is initially copied to the auxiliary memory buffer (`work`).
  - Only the final results are written to `p_out` (always excluding the padded cell positions).

While the previous behavior is ideal, a large memory buffer is needed (`work`). For applications that require memory optimization, diezDecomp offers a memory-lean alternative, which is given by the suroutine: `diezdecomp_transp_execute_generic_nobuf` . This subroutine does not require a memory buffer (`work`), yet it is subject to strong restrictions:

- `p_in` might be overwriten (in-place operations).
- The padded cells covered by `offset6_in/out` might be overwritten (and thus contain left-over data).
- `p_in` and `p_out` must be pointers to memory buffers, with minimum size: `min_size=max(total_size(p_in),total_size(p_out))` .
  - If either `p_in` or `p_out` is shorter than `min_size` , then the respective pointer must start at the beggining of the memory buffer: `p_in/out(0:?,0:?,0:?) -> p_in/out_buf(0:?)` . In this way, Fortran can store extra information at the end of the buffer using the assummed-size arrays (e.g., `p_in(0:*)` or `p_out(0:*)`).

While the previous conditions sound restrictive, DNS/LES solvers typically use transpose operations inside Poisson solvers or parallel tridiagonal solvers (PCR-DTDMA). In both of these subroutines, the previous conditions can be accepted:

- The input array `p_in` contains temporary values obtained from FFT operations or cyclic reduction, which are discarded afterwards.
  - This implies that overwriting `p_in` using in-place operations is acceptable.
- The information in the padded cells of `p_in/out` is outdated after all the operations performed by Poisson solvers or PCR-DTDMA algorithms.
  - Therefore, overwriting such padded cells (`offset6_in/out`) is not an issue.
  - If halo cells are necessary, the halo exchange functions must be called again.
    - Some DNS codes do not require halo cells, but rather use padded cells as auxiliary storage to meet the requirements of libraries, such as `cuDecomp` , `rocFFT` , etc.
      - In these scenarios, the values of the padded cells can always be overwritten after the FFT operations are completed.
- As it is shown in the Fortran example, only a few lines of code are required to modify the declaration of `p_in` or `p_out` , such that they are pointers towards (existing) memory buffers.

Based on the previous discussion, `diezdecomp_transp_execute_generic_nobuf` is a valid alternative for many DNS solvers requiring memory optimization. However, the user is responsible for ensuring that the previous conditions are acceptable nonetheless.

The Fortran code in Example 3 showcases how `diezdecomp_transp_execute_generic_nobuf` can be used to perform the same generic transposes shown in Example 2, obtaining identical results for the (non-padded) cells in `p_out` .

Autotuning for Transpose Operations

diezDecomp features multiple autotuning choices for transpose operations:

- Synchronous vs Asynchronous MPI operations
- Batched vs Simultaneous Data Packing/Unpacking
- For-loop Order for GPU Kernels in Packing/Unpacking Operations
- Intermediate MPI buffer Indexing Order

Each of these options is explained below.

Synchronous vs Asynchronous MPI operations

By default, diezDecomp works with asynchronous MPI operations (`MPI_ISend/IRecv`). However, it is also possible to enable (synchronous) `MPI_Alltoallv` transpose operations.

This is accomplished with:

- Generic API: Activate the flag `allow_alltoallv` in the subroutine `diezdecomp_generic_fill_tr_obj` .
- CaNS API: Enable the global flag `diezdecomp_enable_alltoallv` (CaNS API).

If `MPI_Alltoallv` operations are allowed, diezDecomp will first perform strict checks to determine if the option is compatible with the input/output grid layout for the MPI tasks: ($p_x \times p_y \times p_z$). The compatibility checks involve multiple factors, such as analyzing the (local) order of the MPI tasks in every row/column, and whether information must be transfered in the Cartesian direction orthogonal to the transpose operation. If the `MPI_Alltoallv` operation is rejected by the compatibility checks, diezDecomp will forcefully disable this option (`use_alltoallv=.false.`), and asynchronous MPI operations (`MPI_ISend/IRecv`) will be used instead.

When `MPI_Alltoallv` transfers are permitted (after the compatibility checks), diezDecomp will perform a quick benchmark to check if this is faster than asynchronous MPI transfers (`MPI_ISend/IRecv`).

If the user wishes to force diezDecomp to use `MPI_Alltoallv` operations (regardless of the autotuning results or compatibility checks), the following code can be used:

```
! initialize the transpose descriptor (tr)
call diezdecomp_generic_fill_tr_obj(tr, obj_rank_in, obj_rank_out, sp_in, offset6_in, sp_out, offset6_out, order_in, order_out, order_intermediate, allow_alltoallv, i_a2av, j_a2av, wsize, allow_autotune_reorder)

! ----- force diezDecomp to use mpi_alltoallv -----
tr%use_alltoallv  = .true.  ! enable mpi_alltoallv (warning: this overwrites the compatibility checks)
tr%use_isendirecv = .false. ! disable asynchronous operations (mpi_isend/irecv)
tr%chosen_backend = .true.  ! avoid autotuning
```

The meaning of the last three variables is summarized below:

- `tr%use_isendirecv` :
  - This option enables `MPI_ISend/IRecv` transfers for diezDecomp transposes. Due to the robustness of this approach, all initialization methods for diezDecomp transposes enable this option (by default).
- `tr%use_alltoallv` :
  - Boolean flag to enable `mpi_alltoallv` transposes in diezDecomp. As discussed before, all diezDecomp initialization methods will perform compatibility checks to determine if this option is feasible. If issues are found, `use_alltoallv=.false.` will be forced, even if the flags `allow_alltoallv` (generic API) or `diezdecomp_enable_alltoallv` (CaNS API) were activated in the initialization routine.
- `tr%chosen_backend` :
  - This variable is used by diezDecomp to check whether a communication backend has already been chosen (`MPI_ISend/IRecv` vs. `MPI_Alltoallv`).
    - If both `use_alltoallv` and `use_isendirecv` are active, diezDecomp will perform an autotuning benchmark, and the slowest option will be disabled.
    - Otherwise, if only one MPI option is active (`use_alltoallv` or `use_isendirecv`), diezDecomp will continue using the chosen option.
      - Due to this edge-case behavior, manually changing `chosen_backend` is not strictly necessary.
  - Please note that if the user adds new communication backends to diezDecomp (e.g. NCCL or RCCL), it necessary to disable them too when `chosen_backend` is set to `.false.` .

Batched vs Simultaneous Data Packing/Unpacking

In transpose operations, the main performance bottleneck are typically MPI data transfers. However, other operations can also create significant performance overheads. For instance, MPI subroutines use 1D data buffers; containing all information sent or received by the operation. Therefore, data must be packed into these 1D buffers before sending, and received data must be unpacked afterwards.

For small-scale simulations, one of the most efficient approaches is to launch one GPU kernel to pack or unpack data for every GPU device used as target/destination in the (local) MPI operation. This is called a "batched" mode, since data is processed in small portions.

However, for extreme-scale simulations, batched operations are inefficient, since thousands of GPU kernels can be launched together. To avoid this issue, diezDecomp includes a "simultaneous" data packing/unpacking mode, where only one GPU kernel is launched to handle the entire 1D memory buffer of the MPI transfer. The "simultaneous" operation method is susbtantially more efficient for extreme-scale simulations, and it can also deliver high performance for small-scale simulations.

By default, diezDecomp performs auto-tuning to determine if the "batched" or "simultaneous" mode is more efficient. Packing and unpacking operations are autotuned separately. Usually, "batched" operations are faster for small-scale simulations, whereas "simultaneous" data packing/unpacking is better for extreme-scale simulations. The main reason that the "simultaneous" mode is (slightly) slower for small DNS/LES runs is because it uses an advanced GPU kernel containing more variables, which are simplified in "batched" runs (at the expense of launching multiple kernels).

To disable auto-tuning, the user can manually specify the mode for data packing/unpacking with the following variables:

- `tr%send_autotuned` :
  - Mode for data packing before MPI transfers. (`1` : simultaneous, `2` : batched)
- `tr%recv_autotuned` :
  - Mode for data unpacking after MPI transfers. (`1` : simultaneous, `2` : batched)

For-loop Order for GPU Kernels in Packing/Unpacking Operations

Beyond choosing between "batched" or "simultaneous" modes (for data packing/unpacking), another important factor is the for-loop order of the GPU kernels. While it may sound redundant, for transposes having different input/output array orders (e.g. $y/z/x \rightarrow x/z/y$), choosing the fastest for-loop order for the GPU kernels is not trivial (e.g. $k \rightarrow j \rightarrow i$ or $k \rightarrow i \rightarrow j$). The running times of GPU kernels can change by an order of magnitude, depending on the for-loop order.

To avoid performance issues, diezDecomp performs an autotuning benchmark to identify the optimal for-loop order for its GPU kernels. Each mode (batched vs. simultaneous) and task (packing vs. unpacking) is benchmarked separately to identify the best for-loop order [1].

Based on experience, the for-loop orders of the GPU kernels in diezDecomp can be chosen by the user using the following variables:

- `tr%send_mode_op_batched` :
  - For-loop order for the "batched" mode, in data packing operations (before MPI transfers).
- `tr%send_mode_op_simul` :
  - For-loop order for the "simultaneous" mode, in data packing operations (before MPI transfers).
- `tr%recv_mode_op_batched` :
  - For-loop order for the "batched" mode, in data unpacking operations (after MPI transfers).
- `tr%recv_mode_op_simul` :
  - For-loop order for the "simultaneous" mode, in data unpacking operations (after MPI transfers).

The previously listed variables correspond to integers, with values between [1-6]:

- 1: $i \rightarrow j \rightarrow k$
  - The input array always has a (local) index order: $A(i,j,k)$. Only the for-loop order changes.
  - Please note that MPI 1D data buffers can use another indexing order (e.g. (j,i,k)), which is controlled by the variable `order_intermediate` mentioned in Example 2.
- 2: $i \rightarrow k \rightarrow j$
- 3: $j \rightarrow i \rightarrow k$
- 4: $j \rightarrow k \rightarrow i$
- 5: $k \rightarrow i \rightarrow j$
- 6: $k \rightarrow j \rightarrow i$

In Fortran GPU kernels, for 3D arrays with $A(i,j,k)$ indexing, the for-loop order $k \rightarrow j \rightarrow i$ (number 6) is usually among the fastest, whereas the modes (1-2) are the slowest ($i \rightarrow j \rightarrow k$ and $i \rightarrow k \rightarrow j$). The first two modes (1-2) are slower, because i is the innermost dimension in Fortran, and thus the for-loop order should be reversed. Due to this reason, diezDecomp disables the modes (1-2) during autotuning benchmarks, but the user can enable them if desired.

[1]: Technically, each GPU kernel in the "batched" packing/unpacking mode can be benchmarked separately for its optimal for-loop order. This is relatively simple to accomplish in diezDecomp, but it has been disabled for now. The main technical challenge is that the precision of `MPI_WTIME` is often coarser than the measured running times. Moreover, the relevance of further autotuning is questionable. With the current strategy, the data packing/unpacking operations are typically orders of magnitude faster than MPI operations.

Intermediate MPI buffer Indexing Order

As previously mentioned, the indexing order of the MPI data buffers can also affect the performance of diezDecomp. Due to this reason, users are allowed to perform an auto-tuning benchmark for the best indexing order of the MPI data buffers.

This option corresponds to a nested for-loop, which will perform all the previous (autotuning) benchmarks for each configuration (`order_intermediate`):

- $i \rightarrow j \rightarrow k$
- $i \rightarrow k \rightarrow j$
- $j \rightarrow i \rightarrow k$
- $j \rightarrow k \rightarrow i$
- $k \rightarrow i \rightarrow j$
- $k \rightarrow j \rightarrow i$

As a general rule, this benchmark can be avoided for input/output transpose arrays with simple indexing orders. For instance, if both the input and output arrays have the same index order (e.g. (j,i,k)), using the same format for the MPI 1D buffers is likely a good choice.

In the generic API (`diezdecomp_generic_fill_tr_obj`), the autotuning of the indexing order for the MPI buffers can be activated with the boolean flag: `allow_autotune_reorder` . Similarly, in the CaNS API, the (boolean) global variable `diezdecomp_allow_autotune_reorder` controls whether this type of autotuning is performed.

If autotuning is disabled, the variable `order_intermediate` mentioned in Example 2 will determine the indexing order for the MPI data buffers. Please note that both MPI send and receive 1D data buffers have the same indexing system.

Autotuning Report (Transposes)

So far, this section has presented the autotuning options for diezDecomp transposes. To understand better the impact of different choices, diezDecomp includes the subroutine `diezdecomp_summary_transp_autotuning` , which prints a summary of all autotuned parameters in the code. The transpose object `tr` has internal variables that record the benchmarked times for every tested option. Therefore, no additional benchmarks are performed when `diezdecomp_summary_transp_autotuning` is called.

Additionally, please note that for DNS/LES solvers, there likely exist fixed autotuning parameter choices that deliver high performance in most practical settings:

- Synchronous vs asynchronous MPI transfers.
- Batched vs. simultaneous data packing/unpacking modes.
- Indexing order for MPI data buffers (`order_intermediate`).
- For-loop orders.

Further Information

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