Octopus: Functions on the mesh

Martin Lüders

Octopus Course 2023, MPSD Hamburg

Mesh functions

Mesh functions:

- position dependent quantities are stored as so-called mesh functions.
- Examples:
  - density
    `rho(1:mesh%np, 1:st%d%nspin)`
    (no ghost points needed here)
  - Hartree potential `hm%vhartree(1:mesh%np_part)`
  - vector potential
    `hm%a_ind(1:mesh%np_part, 1:space%dim)`
- wave functions are stored differently $\longrightarrow$ batches

Mesh functions

Operations on mesh functions:

- local operations: point-wise operation, simple loop (consider BLAS/LAPACK)
- integrations: summation in each domain and reduction over domains
- derivatives: need to consider ghost and boundary points

Mesh functions

Pre-defined operations on mesh functions:

- Integrations (see e.g. `mesh_function_oct_m`)

  dot product `X(mf_dotp)(mesh, f1, f2, reduce, dotu, np)`

  norm `X(mf_nrm2)(mesh, ff, reduce)`

- Derivatives (see e.g. `derivatives_oct_m`)

  Laplacian
  ```
  X(derivatives_lapl)(der, ff, op_ff,
  ghost_update, set_bc, factor)
  ```

  gradient
  ```
  X(derivatives_grad)(der, ff, op_ff,
  ghost_update, set_bc)
  ```

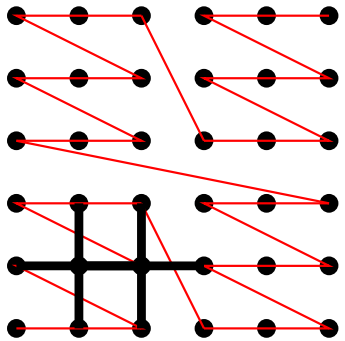
However: We are trying to use batches wherever possible.

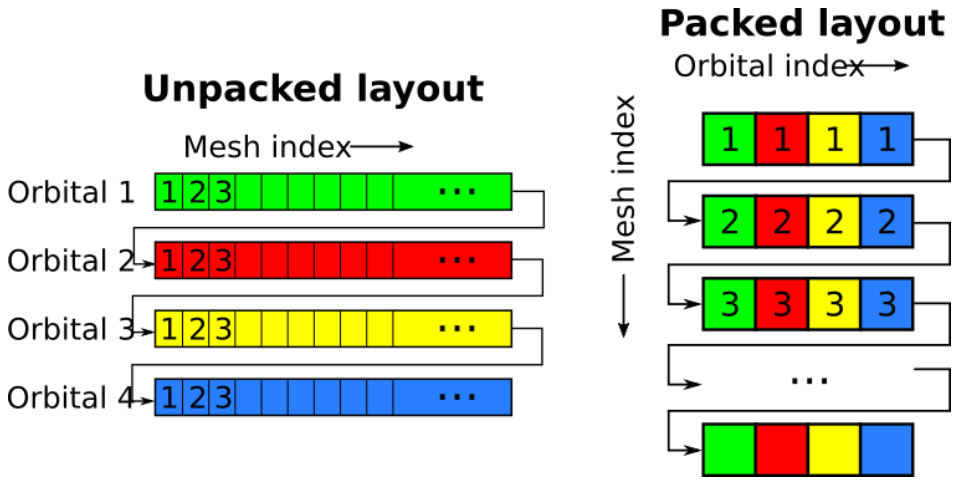
Batches

Batches of mesh functions

Motivation

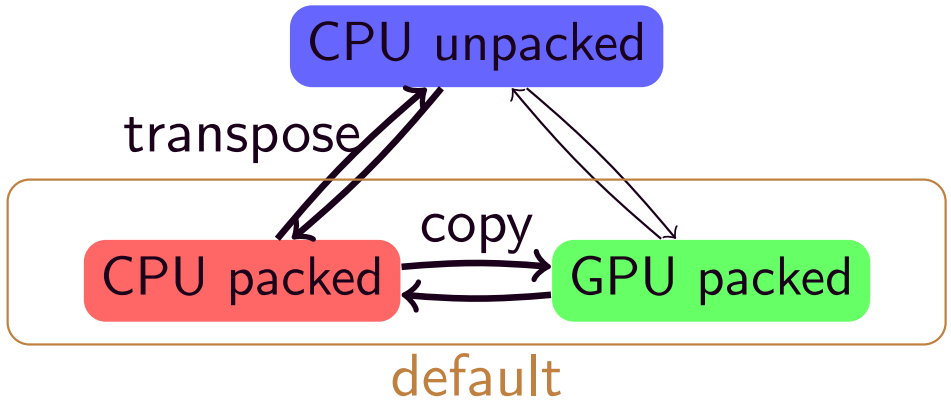
- stencil depends on mesh point
- often one has to operate on many mesh functions at once (e.g. wave functions)
- more effective to swap mesh index and function index
- 'packed form': fast index is now over states.
- Same stencil for every group.





Batch status

- batches can have three states:
- transitions: (mostly under the hood)
  - `psib%do_pack()`,
  - `psib%do_unpack()`
- default is packed or GPU packed (if device enabled)



Batches

excerpt from `batch_t`:

```
! unpacked variables; linear variables are pointers with different shapes
FLOAT, pointer, contiguous, public :: dff(:, :, :)     !< (1:np,1:dim, 1:nst)
CMPLX, pointer, contiguous, public :: zff(:, :, :)     !< (1:np,1:dim, 1:nst)
FLOAT, pointer, contiguous, public :: dff_linear(:, :) !< (1:np,1:nst_linear)
CMPLX, pointer, contiguous, public :: zff_linear(:, :) !< (1:np,1:nst_linear)

! packed variables; only rank-2 arrays due to padding to powers of 2
FLOAT, pointer, contiguous, public :: dff_pack(:, :)   !< (1:nst_linear,1:np)
CMPLX, pointer, contiguous, public :: zff_pack(:, :)   !< (1:nst_linear,1:np)

type(accel_mem_t),          public :: ff_device   !< pointer to device memory
```

Note:

- the unpacked memory can be a pointer to some externally provided array of mesh function.
- manipulating the batch means that this memory is manipulated.
- calls to `batch_end()` makes sure that the memory is in sync, i.e. the batch will be unpacked and/or copied back from the GPU memory.

Manipulating batches

- Usually no low-level access needed.
- Many mathematical operations are provided:
  - `batch_ops_oct_m`:
    operations on batches which do not require knowing the mesh and parallelization (no reduction), local operations
    Batch equivalent of BLAS/Lapack calls (axpy, scal,...)
  - `mesh_batch_oct_m`:
    global operations like dot products
    Batch equivalent of `mesh_function_oct_m` routines
  - `derivatives_oct_m`:
    batch versions of the derivative routines

Wave functions

Eletronic wave functions functions

Electrons

- The top level object for electrons is `electrons_t`.
  - System in the multisystem framework
  - Contains: hamiltonian, states, td, scf, k-points, etc.
- The states are stored in `type(states_elec_t) :: st`
  - Top level object for the states and related properties
  - Contains:
    - wavefunctions (in `type(states_elec_group_t) :: group`)
    - eigenvalues, occupations
    - density, currents
    - etc.
  - the module provides functions for:
    - allocating, initializing, copying, freeing, states
    - distributing states
    - locating a state (state and k-point parallelization)
    - getting and setting functions in a state

Electrons

- wave function groups (in `states_elec_group_t`):
  - Contains states and information how they are distributed
  - Blocks of states are in `type(wfs_elec_t) :: psib`

Batches of wave functions

- The type `wfs_elec_t` extends the `batch_t` type
- It inherits from `batch_t`:
  - `nst`: number of states in the batch
  - `dim`: spin dimension
    (1 for non magnetic or collinear magnetism, 2 for spinors)
  - `nst_linear`: number of mesh functions in the batch (nst*dim)
  - mappings, etc.
- It adds
  - k-point index
    each wave function in the batch has the same k-point.
  - information whether the wave functions carry the phase

Working with wave functions

- How to access a wave function from the `st` object? `call states_elec_get_state(st, mesh, ist, ik, psi)`

  `st` `states_elec_t` type object

  `mesh` `mesh_t` type object

  `ist` state index

  `ik` k-point / spin index

  `psi` the wave function (`(1:mesh%np, 1:st%d%dim)`)

- avoid manipulating the function directly
  - use mesh functions (blas, lapack, etc.)
  - if possible, manipulate the batches!

The phase in Octopus

- In periodic systems, the wave function carries a phase: $\Psi_{n,\mathbf{k}}(\mathbf{r}) = u_{n,\mathbf{k}}(\mathbf{r}) e^{i(\mathbf{A}+\mathbf{k})\mathbf{r}}$
- Octopus only stores the periodic part (for easier treatment of boundary points)
- The phase has to be applied before operators act on wave functions
- The phase needs to be removed afterwards

Operators and obervables

Calculating expectation values:

- operators can be expressed in terms of defined math operations
- many terms already implemented in the Hamiltonian
- usually no need to touch low level routines

Let's look at some code: contributions to the total energy (`electrons/energy_calc.F90`)

```
subroutine energy_calc_total(namespace, space, hm, gr, st, iunit, full)

FLOAT function X(energy_calc_electronic)(namespace, hm, der, st, terms) result(energy)

subroutine X(calculate_expectation_values)(namespace, hm, der, st, eigen, terms)
```

Operators and obervables

```
subroutine energy_calc_total(namespace, space, hm, gr, st, iunit, full)

type(namespace_t),         intent(in)    :: namespace
type(space_t),             intent(in)    :: space
type(hamiltonian_elec_t), intent(inout) :: hm
type(grid_t),              intent(in)    :: gr
type(states_elec_t),       intent(inout) :: st
integer, optional,         intent(in)    :: iunit
logical, optional,         intent(in)    :: full
  ...
hm%energy%eigenvalues = states_elec_eigenvalues_sum(st)

if (full_ .or. hm%theory_level == HARTREE &
          .or. hm%theory_level == HARTREE_FOCK &
          .or. hm%theory_level == GENERALIZED_KOHN_SHAM_DFT) then

 if (states_are_real(st)) then
  hm%energy%kinetic      = denergy_calc_electronic(namespace, hm, &
                               gr%der, st, terms=TERM_KINETIC)
  hm%energy%extern_local = denergy_calc_electronic(namespace, hm, &
                               gr%der, st, terms=TERM_LOCAL_EXTERNAL)
  ...
 else
  ... ! same with z prefix
 end if

end if
```

Operators and obervables

```
FLOAT function X(energy_calc_electronic)(namespace, hm, der, st, terms) result(energy)

  type(namespace_t),         intent(in)    :: namespace
  type(hamiltonian_elec_t), intent(in)    :: hm
  type(derivatives_t),       intent(in)    :: der
  type(states_elec_t),       intent(inout) :: st
  integer,                   intent(in)    :: terms

  R_TYPE, allocatable  :: tt(:, :)

  PUSH_SUB(X(energy_calc_electronic))

  SAFE_ALLOCATE(tt(st%st_start:st%st_end, st%d%kpt%start:st%d%kpt%end))

  call X(calculate_expectation_values)(namespace, hm, der, st, tt, terms = terms)

  energy = states_elec_eigenvalues_sum(st, TOFLOAT(tt))

  SAFE_DEALLOCATE_A(tt)
  POP_SUB(X(energy_calc_electronic))

end function X(energy_calc_electronic)
```

Operators and obervables

```
subroutine X(calculate_expectation_values)(namespace, hm, der, st, eigen, terms)

  type(namespace_t),         intent(in)    :: namespace
  type(hamiltonian_elec_t), intent(in)    :: hm
  type(derivatives_t),       intent(in)    :: der
  type(states_elec_t),       intent(inout) :: st
  R_TYPE,                    intent(out)   :: eigen(st%st_start:, st%d%kpt%start:)
  integer, optional,         intent(in)    :: terms

  integer :: ik, minst, maxst, ib
  type(wfs_elec_t) :: hpsib

  do ik = st%d%kpt%start, st%d%kpt%end
    do ib = st%group%block_start, st%group%block_end
      minst = states_elec_block_min(st, ib)
      maxst = states_elec_block_max(st, ib)
      call st%group%psib(ib, ik)%copy_to(hpsib)
      call X(hamiltonian_elec_apply_batch)(hm, namespace, der%mesh, &
                                           st%group%psib(ib, ik), hpsib, terms = terms)
      call X(mesh_batch_dotp_vector)(der%mesh, st%group%psib(ib, ik), hpsib, &
                                     eigen(minst:maxst, ik), reduce = .false.)
      call hpsib%end()
    end do
  end do

  if (der%mesh%parallel_in_domains) call der%mesh%allreduce(&
    eigen(st%st_start:st%st_end, st%d%kpt%start:st%d%kpt%end))

end subroutine X(calculate_expectation_values)
```

Operators and obervables

in `hamiltonian/hamiltonian_elec_inc.F90`:

```
subroutine X(hamiltonian_elec_apply_batch) (hm, namespace, mesh, psib, hpsib, terms, set_bc)

  ...

  ! apply the local potential
  if (bitand(TERM_LOCAL_POTENTIAL, terms_) /= 0) then
    call hm%hm_base%X(calc_local)(mesh, hm%d, hm%d%get_spin_index(psib%ik), epsib, hpsib)
  else if (bitand(TERM_LOCAL_EXTERNAL, terms_) /= 0) then
    call X(hamiltonian_elec_external)(hm, mesh, epsib, hpsib)
  end if

  ...

  POP_SUB(X(hamiltonian_elec_apply_batch))
  call profiling_out(prof_hamiltonian)

end subroutine X(hamiltonian_elec_apply_batch)
```

Operators and obervables

in `hamiltonian/hamiltonian_elec_inc.F90`:

```
subroutine X(hamiltonian_elec_apply_batch) (hm, namespace, mesh, psib, hpsib, terms, set_

  ...

  if (bitand(TERM_KINETIC, terms_) /= 0) then
    ASSERT(associated(hm%hm_base%kinetic))
    call profiling_in(prof_kinetic_start, TOSTRING(X(KINETIC_START)))
    call X(derivatives_batch_start)(hm%hm_base%kinetic, hm%der, epsib, hpsib, handle, &
                                    set_bc = .false., factor = -M_HALF/hm%mass)
    call profiling_out(prof_kinetic_start)
  end if

  ...

  if (bitand(TERM_KINETIC, terms_) /= 0) then
    call profiling_in(prof_kinetic_finish, TOSTRING(X(KINETIC_FINISH)))
    call X(derivatives_batch_finish)(handle)
    call profiling_out(prof_kinetic_finish)
  else
    call batch_set_zero(hpsib)
  end if
```

split in start and finish routine to enable other operations during communication.

Other examples

other noteworthy tasks in `hamiltonian/hamiltonian_elec_inc.F90`:

- packing and unpacking of batches, if required
- application of phase
- application of boundary conditions

Another example to look at: `eigen_chebyshev_inc.F90`

- utilizes only batch functions
- automatically works on GPU