MAX PLANCK
COMPUTING & DATA FACILITY

MAX PLANCK INSTITUTE
FOR THE STRUCTURE AND DYNAMICS OF MATTER

Parallelization and performance: how to make the Octopus swim fast

Nicolas Tancogne-Dejean (MPSD)
Sebastian Ohlmann (MPCDF)

Octopus advanced course, 23.9.2021

Outline

- Optimization strategy, profiling
- Techniques for efficient programming
- Parallelization
- Mesh functions
- Batches

Outline

- **Optimization strategy, profiling**
- Techniques for efficient programming
- Parallelization
- Mesh functions
- Batches

Optimization strategy

- Optimize in serial before going parallel!
- Otherwise: scaling inefficient code
- Iterative procedure
  - Profile: where is the bottleneck?
  - Improve that bottleneck

Profiling

- Measure specified performance metrics for different parts of the code
- Metrics: **time spent**, GFLOPS, memory, …
- Code parts: **functions**, loops, source lines, …
- Critical step: understand code behavior to focus optimization efforts
- “Premature optimization is the root of all evil” (Donald Knuth)
- Pareto rule: “80% of the gains generally come from focusing on 20% of the code”

Profiling tools

- First step: internal profiling
  → time spent in functions
- likwid: FLOPS, memory bandwidth, … for functions
- Intel vtune: time and other metrics on loop level
- Advisor: roof line metrics on loop level
- Nvidia Nsight systems: GPU profiling, data transfers, kernel launches

Internal profiling: usage

- Set input variable ProfilingMode = prof_time
- Output: profiling/time.000000
- Contains timings for regions in the code
- Self-time and cumulative time (ordered by self-time)
- With ProfilingAllNodes = yes: one file per MPI process
- Profiling can be different on different processes due to load imbalance

Example output

		CUMULATIVE TIME						SELF TIME				
TAG	NUM_CALLS	TOTAL_TIME	TIME_PER_CALL	MIN_TIME	MFLOPS	MBYTES/S	%TIME	TOTAL_TIME	TIME_PER_CALL	MFLOPS	MBYTES/S	%TIME
zNL_OPERATOR_BATCH	4060	3.258858	0.000803	0.000209	2591.9	0.0	35.3	3.258858	0.000803	2591.9	0.0	35.3
PS_FILTER	4	1.889388	0.472347	0.327342	0.0	0.0	20.5	1.889388	0.472347	0.0	0.0	20.5
SG_PCONV	101	0.954465	0.009450	0.009319	0.0	0.0	10.3	0.949822	0.009404	0.0	0.0	10.3
zVLPSI	2030	0.619451	0.000305	0.000166	563.1	1223.1	6.7	0.619451	0.000305	563.1	1223.1	6.7
zSET_BC	2030	0.365790	0.000180	0.000098	0.0	0.0	4.0	0.365790	0.000180	0.0	0.0	4.0
MESH_INIT	3	0.357447	0.119149	0.000002	0.0	0.0	3.9	0.344014	0.114671	0.0	0.0	3.7
zGHOST_UPDATE_START	2030	0.264002	0.000130	0.000069	0.0	0.0	2.9	0.259105	0.000128	0.0	0.0	2.8
CALC_DENSITY	505	0.222086	0.000440	0.000232	0.0	0.0	2.4	0.222086	0.000440	0.0	0.0	2.4
ELECTRONS_CONSTRUCTOR	1	0.383280	0.383280	0.383280	0.0	0.0	4.2	0.208536	0.208536	0.0	0.0	2.3
zPROJ_MAT_SCATTER	12180	0.159651	0.000013	0.000002	73.0	0.0	1.7	0.159651	0.000013	73.0	0.0	1.7
COMPLETE_RUN	1	9.226912	9.226912	9.226912	1051.6	84.5	100.0	0.144164	0.144164	0.0	0.0	1.6
HAMILTONIAN_ELEC_INIT	1	2.000087	2.000087	2.000087	0.0	0.0	21.7	0.110699	0.110699	0.0	0.0	1.2
zVNLPSI_MAT_BRA	2030	0.107823	0.000053	0.000034	1009.6	0.0	1.2	0.107823	0.000053	1009.6	0.0	1.2
LIBXC	202	0.073882	0.000366	0.000228	0.0	0.0	0.8	0.073882	0.000366	0.0	0.0	0.8
zVNLPSI_MAT_REDUCE	2030	0.065831	0.000032	0.000004	0.0	0.0	0.7	0.065831	0.000032	0.0	0.0	0.7
BLAS_AXPY_4	3000	0.064741	0.000022	0.000008	5971.5	0.0	0.7	0.064741	0.000022	5971.5	0.0	0.7
zGHOST_UPDATE_WAIT	2030	0.043132	0.000021	0.000011	0.0	0.0	0.5	0.043132	0.000021	0.0	0.0	0.5
BATCH_COPY_DATA_TO	1750	0.042083	0.000024	0.000009	0.0	0.0	0.5	0.042083	0.000024	0.0	0.0	0.5
zVNLPSI_MAT_KET	2030	0.248724	0.000123	0.000058	327.8	0.0	2.7	0.023242	0.000011	3007.3	0.0	0.3
dCUBE_TO_MESH	101	0.020717	0.000205	0.000191	0.0	443.9	0.2	0.020717	0.000205	0.0	443.9	0.2
XC_LOCAL	101	0.093789	0.000929	0.000875	12.8	0.0	1.0	0.019675	0.000195	0.0	0.0	0.2
zHAMILTONIAN	2030	4.943666	0.002435	0.001494	1817.6	153.3	53.6	0.018308	0.000009	0.0	0.0	0.2
EXP_TAYLOR_BATCH	500	5.000877	0.010002	0.006267	1912.0	149.3	54.2	0.016696	0.000033	0.0	0.0	0.2
dMESH_TO_CUBE	101	0.024494	0.000243	0.000231	0.0	375.4	0.3	0.015987	0.000158	0.0	575.2	0.2
POISSON_SOLVE	101	1.017069	0.010070	0.009940	0.0	18.1	11.0	0.015378	0.000152	0.0	0.0	0.2

Internal profiling: implementation

- Define object and call profiling_in/profiling_out:
  - use profiling_oct_m
  - …
  - subroutine …
  - type(profile_t), save :: exp_prof
  - call profiling_in(exp_prof, "EXPONENTIAL")
  - …
  - call profiling_out(exp_prof)
  - end subroutine

More internal profiling

- More options for ProfilingMode
  - **prof_io**: count number of file open/close operations
  - **prof_mem**: summary on memory usage and largest array
  - **prof_mem_full**: log of every allocation and deallocation

Profiling: tips & tricks

- Internal profiling (prof_time) can be always enabled
- Negligible overhead
- Data available for past runs → quick check possible
- For TD runs: TIME_STEP for full time steps
- For GS runs: SCF_CYCLE for full iterations

Profiling on GPUs

- Compile with CUDA and NVTX support:
  ./configure --enable-cuda --enable-nvtx …
- Enable profiling (ProfilingMode = prof_time)
- Install Nsight systems (or use nsight_systems/2021 module on MPCDF systems)
- Run Nsight:
  - nsys profile -t cuda,nvtx,mpi srun -n 2 octopus
- Will create reportXX.qdrep
- Open with GUI (nsys-ui) – either with X forwarding or on local PC
- Analyze timeline (NVTX regions, kernel launches, data transfers)

Profiling for parallel runs

- Default: profiling written by rank 0
- Possible problem: load imbalance
  - Different timings on different ranks
  - Might lead to wrong conclusions
- Set ProfilingAllNodes = yes
  - Writes out profiling for all ranks
  - Comparison possible

Outline

- Optimization strategy, profiling
- **Techniques for efficient programming**
- Parallelization
- Mesh functions
- Batches

Efficient code

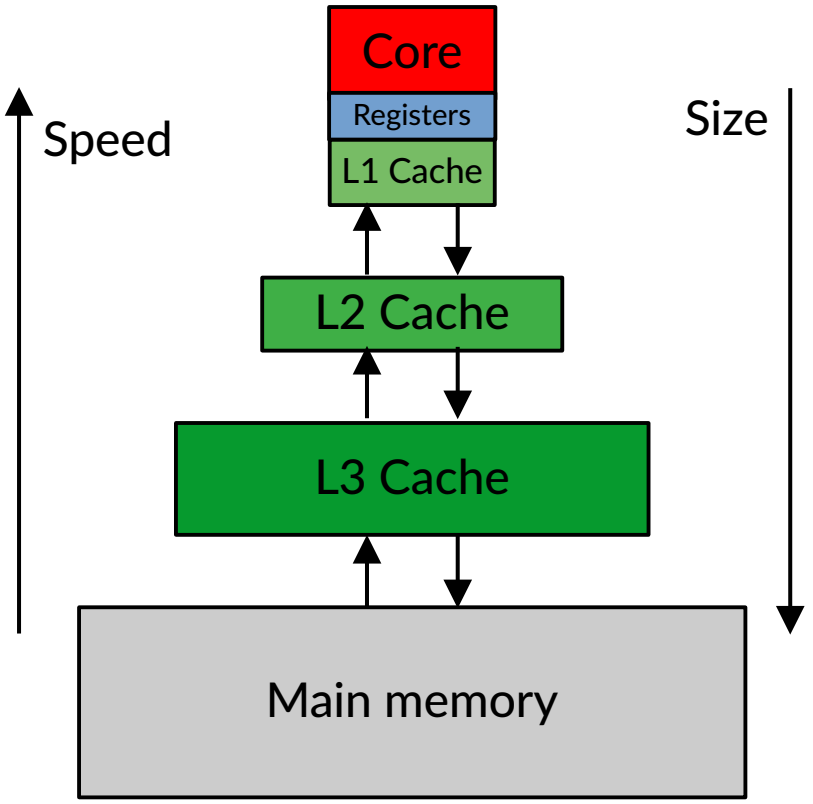
- For implementing a feature:
  - Use operations already implemented for batches or mesh functions (e.g. scaling, integral, …; see later)
  - Use blas/lapack functions
  - Implement loops yourself

Efficient loops

- Most important: memory access pattern
- Memory access: linear → best use of caches
- Important:
  - Layout of array in memory
  - Order of loops

Memory access

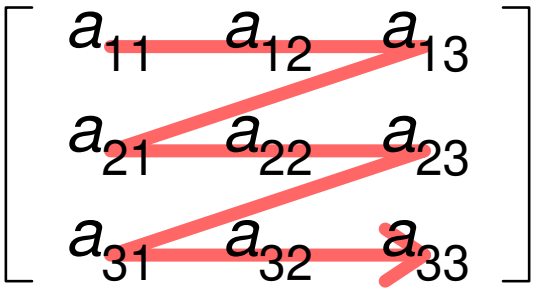
- Memory much slower than CPU → often a bottleneck
- Mitigation: hierarchy
- Cache is filled in small chunks
- Most performance:
  - Linear access
  - Reuse memory



Memory layout of arrays

- C: row-major layout
- Fortran: column-major layout
  - First index changes fastest
  - Last index changes most slowly
  - Linear access: innermost loop over first index!

Row-major order



Column-major order

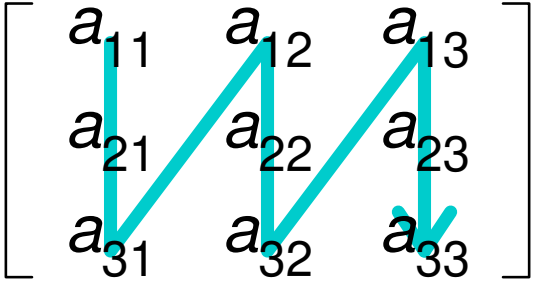


Image source: Wikipedia, Author: Cmglee

Example code: phase

- First dimension: ii
  - Fastest index
  - Inner loop
- Second dimension: ip
  - Slowest index
  - Outer loop

```
do ip = 1, min(mesh%np, np)
  phase = this%phase(ip, psib%ik)
  do ii = 1, psib%nst_linear
    psib%zff_pack(ii, ip) = phase * &
                    src_%zff_pack(ii, ip)
  end do
end do
```

BLAS/LAPACK functions

- Interfaces in
  - math/lalg_basic.F90
    - e.g. lalg_axpy, lalg_nrm2, lalg_gemm
  - math/lalg_adv.F90
    - e.g. lalg_cholesky, lalg_eigensolve, lalg_determinant
  - For different dimensions of arrays
- Use efficient BLAS/LAPACK implementation
  - MKL, OpenBLAS

BLAS example

- Example function: lalg_axpy
- Compute y = a*x + y
- Example in CG eigensolver:
  - psi, cg: 2D arrays (mesh%np_part, st%d%dim)
  - **call lalg_axpy(mesh%np, st%d%dim, -norma, psi, cg)**
  - Compute cg = -norma*psi + cg
  - Corresponds to orthogonalization

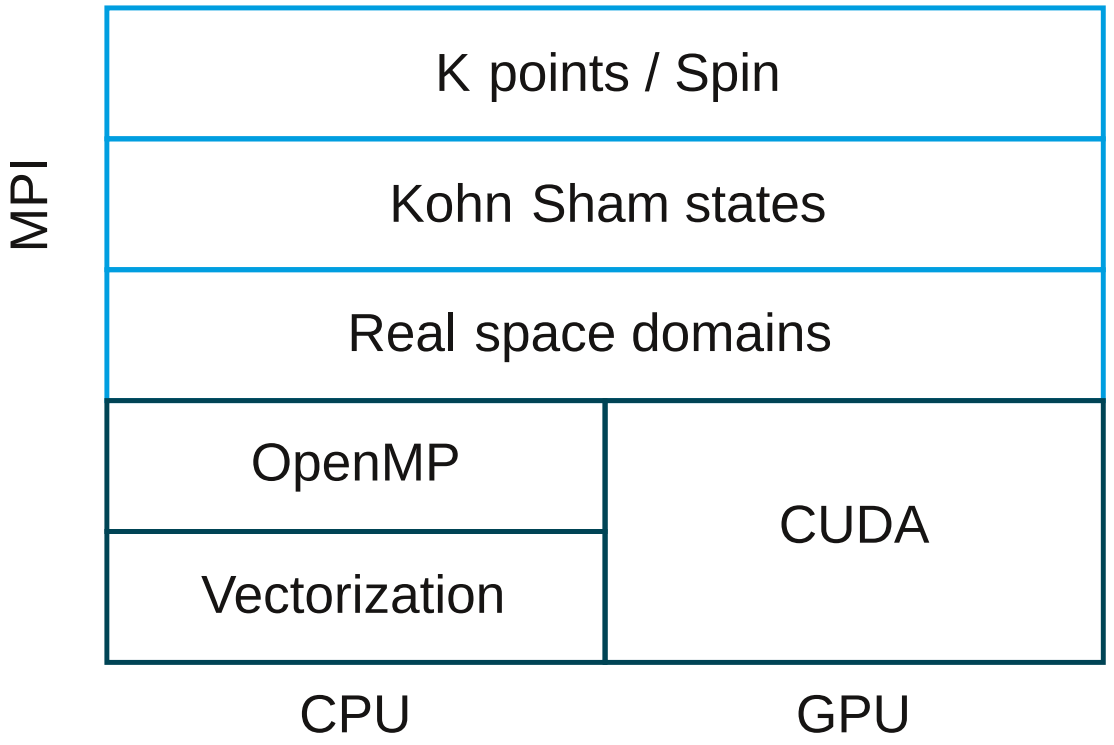
High-level operations

- Most efficient: use batch operations
  - Basic operations implemented also on GPUs
  - Use vectorization on CPUs
- Next level: mesh functions
- More details later

Outline

- Optimization strategy, profiling
- Techniques for efficient programming
- **Parallelization**
- Mesh functions
- Batches

Parallelization strategies



Guidelines

- K points: min. 1 k point per process
- States: min. 4-8 states per process
- K points and states should be balanced
- States: most efficient is multiple of StatesBlockSize (number of states in a batch)
  - CPUs: 4
  - GPUs: 32
- Domains: ratio ghost/local points <25%

Parallelization in the code

- Distribution of k points:
  ik from st%d%kpt%start to st%d%kpt%end
- Distribution of states:
  ist from st%st_start to st%st_end
- Wavefunctions: in groups of states (batches)
  ib from st%group%block_start to st%group%block_end
- Access certain batch: st%group%psib(ik, ib)
- Domains:
  - Local number of points: np, np_part (includes ghost + boundary points)
  - Global number of points: np_global, np_part_global

Loop over k points and states

- Example: subroutine states_elec_set_zero

```
do iqn = st%d%kpt%start, st%d%kpt%end
  do ib = st%group%block_start, st%group%block_end
    call batch_set_zero(st%group%psib(ib, iqn))
  end do
end do
```

- Loops over local part of states → enables parallelization

Loop over domains

- Should be rarely needed
- Rather use BLAS/LAPACK or batch functions
- Simply loop from 1 to mesh%np
- Points from np to np_part: ghost and boundary points, should normally not be touched
- For certain operations, reduction necessary (e.g. integrals, sums, …) → see mesh functions

Hints

- Take parallelization into account from the beginning!
- Easier than later modification
- Most important:
  - Distribution of data
  - Work on locally available data (→ correct loop boundaries)

Outline

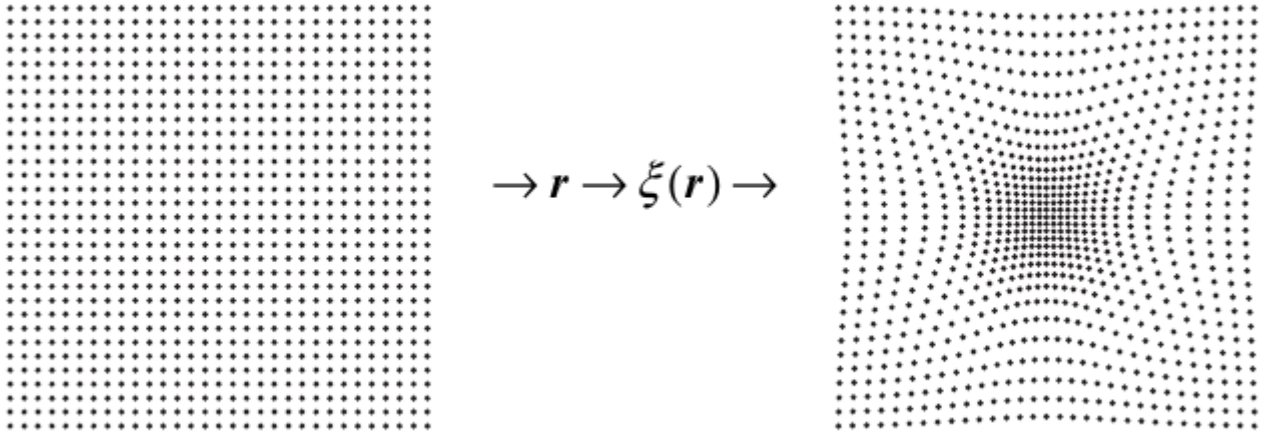
- Optimization strategy, profiling
- Techniques for efficient programming
- Parallelization
- **Mesh functions**
- Batches

Scope

- We want to use Octopus efficiently:
  - calculation should be fast
  - code should not use too much memory
  - minimal IO and communications
  - use resources efficiently: multi cores, GPUs, …
  - simple, maintainable code

A non trivial task

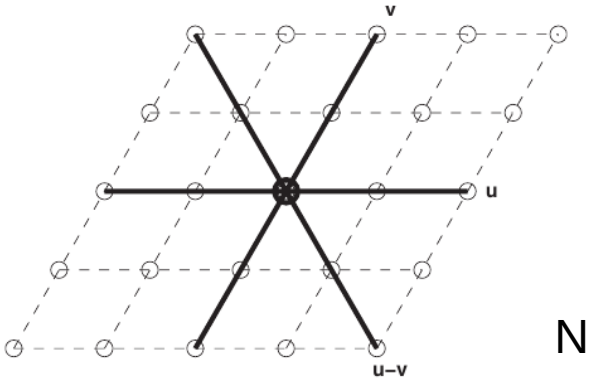
- Mesh can be non uniform, e.g., curvilinear mesh
  - Affects weights for finite differences, integrals, dot products, ...



A. Castro *et al.*, phys. stat. sol. (b) 243, 11 (2006)

A non trivial task

- Mesh can be non uniform, e.g., curvilinear mesh
  - Affects weights for finite differences, integrals, dot products, ...
- Space can be generated along non-orthogonal lattice vectors
  - Affects derivatives and observables, e.g. forces, current, ...



Natan et al., PRB 78, 075109 (2008)

A non trivial task

- Mesh can be non uniform, e.g., curvilinear mesh
  - Affects weights for finite differences, integrals, dot products, ...
- Space can be generated along non-orthogonal lattice vectors
  - Affects derivatives and observables, e.g. forces, current, ...
- Data can be on CPU or GPU
  - Copies might be needed to access data
- Support of OpenMP/MPI
  - Implies communications like reductions over threads/tasks
- Multiple dimensions (1D, 2D, 3D, 4D, ...)

What we ~~should~~ must not do !

Taken from *src/hamiltonian/kb_projector_inc.F90*

```
do idim = 1, dim
  do ic = 1, kb_p%n_c
    do is = 1, ns
      uvpsi(idim, ic) = uvpsi(idim, ic) + psi(is, idim)*kb_p%p(is, ic)
    end do
  end do
end do
```

What we ~~should~~ not do ! must

Taken from *src/hamiltonian/kb_projector_inc.F90*

```
do idim = 1, dim
  do ic = 1, kb_p%n_c
    do is = 1, ns
      uvpsi(idim, ic) = uvpsi(idim, ic) + psi(is, idim)*kb_p%p(is, ic)
    end do
  end do
end do
```

Number of grid points

Functions on the grid
“mesh functions”

What we ~~should~~ not do ! must

Taken from *src/hamiltonian/kb_projector_inc.F90*

```
do idim = 1, dim
  do ic = 1, kb_p%n_c
    do is = 1, ns
      uvpsi(idim, ic) = uvpsi(idim, ic) + psi(is, idim)*kb_p%p(is, ic)
    end do
  end do
end do
```

Number of grid points

Functions on the grid
“mesh functions”

What do we compute here:
- Compute a series of dot product here for each projector (labeled by ic) resolved per spinor dimension (idim)

What is bad here:
- No BLAS call
- No OpenMP support
- No GPU support
- Code specific to uniform grids. The curvilinear case is not supported

Where do I find relevant routines?

Different levels:

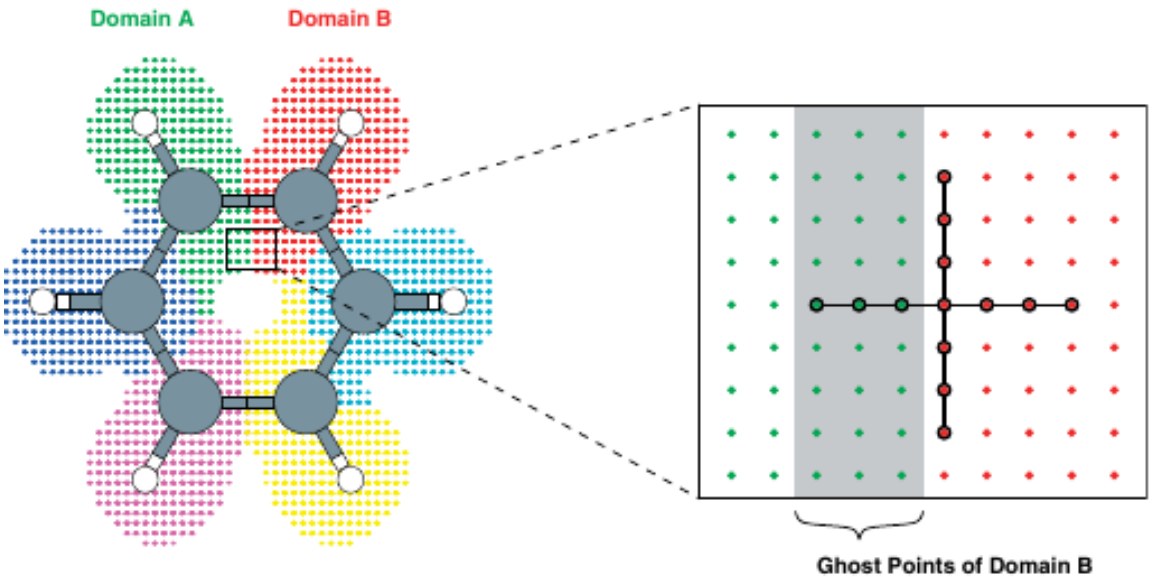
- In the grid folder: how to manipulate individual mesh functions (or batches, see later)
  - Dot product, integral, norm, linear algebra like BLAS axpy,...
- In the states folder: manipulate all states at once
  - Randomization of states, orthogonalization,...
- In the electrons folder: same as states, but needs to know the Hamiltonian
  - Subspace diagonalization

Wavefunctions and data in Octopus

- Two possible cases:
  - “**mesh functions**”: one dimensional arrays
  - “**batches**”: collections of mesh functions packed together in memory

Mesh functions

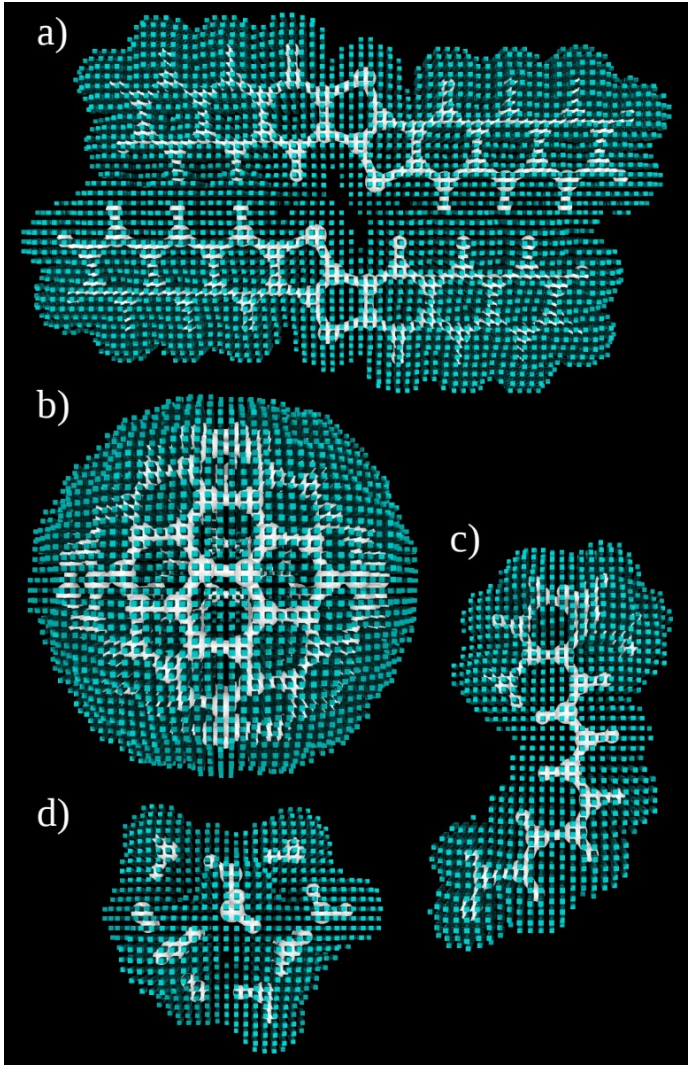
- Contains weight of the function *f* evaluated at the grid point r_i: *array(i)=f(r_i)*
- The grid is divided in real-space domains



A. Castro *et al.*, phys. stat. sol. (b) 243, 11 (2006)

Data layout

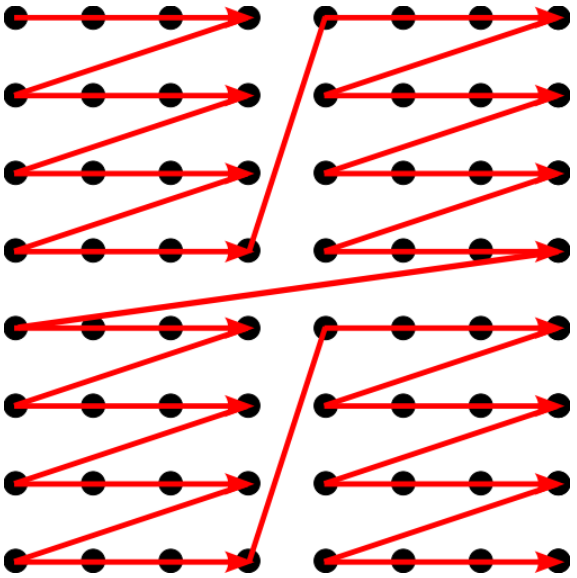
- Complicated shape possible, e.g. molecules



X. Andrade & A. Aspuru-Guzik, J. Chem. Theory Comput. (2013), 9, 10, 4360-4373

Data layout

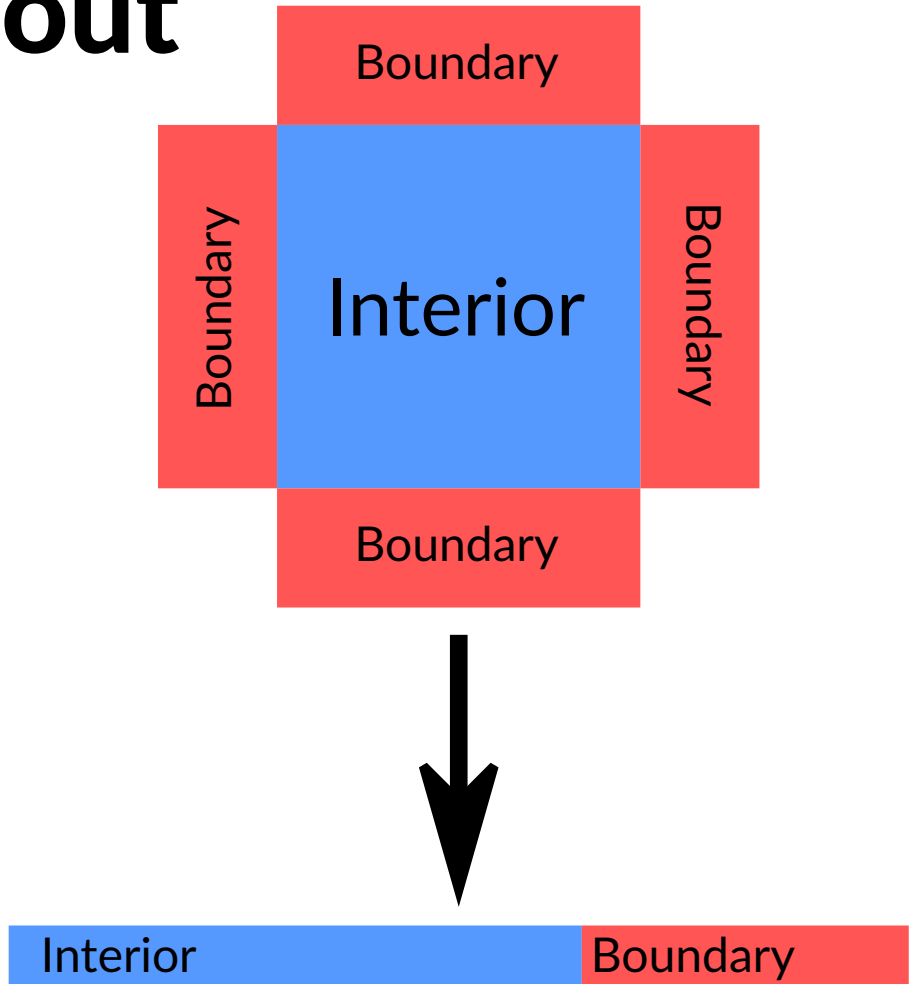
- Complicated shape possible, e.g. molecules
- Cache-aware mapping to 1D array



X. Andrade & A. Aspuru-Guzik, J. Chem. Theory Comput. (2013), 9, 10, 4360-4373

Data layout

- Complicated shape possible, e.g. molecules
- Cache-aware mapping to 1D array
- 1D data layout: 2 blocks
  - Interior points
  - Boundary/ghost points



Mesh functions

- Contains weight of the function *f* evaluated at the grid point r_i : $array(i)=f(r_i)$
- The grid is divided in real-space domains
- Locally, we have np points

- We also have ghost points:
  - From other domains
- We also have boundary points:
  - Describing boundaries of the simulation box

Mesh functions

- If we need to perform derivatives, we have locally *np_part* points

- Usually data stored from *np+1* to *np_part* don’t need to be manipulated: done automatically when performing derivatives

np versus np_part

The question to ask yourself:

do I need to compute derivatives ?

- If no, the array should be of size *np*
- Else, use *np_part*

Important for not using too much memory

Reduced communication and transfers from/to GPU

Much fast, less operations are performed

Example: In bulk Si, primitive cell, we have

- *np=2744*
- *np_part=9192*

How to manipulate mesh functions?

- Let's assume that you know a mesh function f on the grid (and its friends g, h, ...)
- Octopus provide basic "safe" operations

  src/grid/mesh_function.F90

  e.g. X(mf_integrate), X(mf_dotp), X(mf_nrm2)

- Safe for uniform and non-uniform meshes
- Support OpenMP and MPI
- Internally use BLAS when possible

How to manipulate mesh functions?

- Let's assume that you know a mesh function f on the grid (and its friends g, h, …)
- Octopus provide access to BLAS/LAPACK calls

  src/math/lalg_basic.F90 – src/math/lalg_adv.F90

  e.g. lalg_axpy, lalg_scal, …

Must only be used for local operators, not for global operations (dot products, norms, integrals, …).

How to get a mesh function?

If you know the states_elec_t object:

call states_elec_get_state(st, mesh, ist, ik, psi)

st: states_elec_t object

mesh: mesh_t object

ist: state index

ik: k-point/spin index

psi(1:mesh%np, 1:st%dim): a wavefunction (or Pauli spinor)

Returns only *np* points; *np+1* to *np_part* are not set

How to get a mesh function?

If you know a wfs_elec_t object (or batch_t):

call batch_get_state(psib, ist, np, psi)

psib: batch_t or wfs_elec_t object

ist: index of the state in the batch. Not the state index ! Goes from 1 to *psib%nst*.

np: number of points requested. Usually *mesh%np*, sometimes *mesh%np_part*.

psi(1:mesh%np, 1:st%dim): a wavefunction (or Pauli spinor)

The batch caries the information of the state/k-point indices

How to set a mesh function?

Once you have finished manipulating the mesh function:

- batch_set_state
- states_elec_set_state

Warning: every call to get_state/set_state implies a copy/transfer.

Needs to be avoided → see batch manipulation

Example 1: Gram-Schimdt orthonormalization

The algorithm (from wikipedia):

$$
\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \operatorname{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}
$$

Where

$$
\operatorname{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\langle \mathbf{u}, \mathbf{v} \rangle}{\langle \mathbf{u}, \mathbf{u} \rangle} \mathbf{u},
$$

Basic operations: dot product, norm, y = a*x+y

Example 1: Gram-Schimdt orthonormalization

Taken from *src/states/states_elec_calc_inc.F90*

```
call states_elec_get_state(st, mesh, ist, ik, psii)

! calculate the projections first with the same vector
do jst = 1, ist - 1
  call states_elec_get_state(st, mesh, jst, ik, psij)
  aa(jst) = X(mf_dotp)(mesh, st%d%dim, psij, psii, reduce = .false.)
end do

if (mesh%parallel_in_domains .and. ist > 1) call mesh%allreduce(aa, dim = ist - 1)
! then subtract the projections
do jst = 1, ist - 1
  call states_elec_get_state(st, mesh, jst, ik, psij)
  do idim = 1, st%d%dim
    call lalg_axpy(mesh%np, -aa(jst), psij(:, idim), psii(:, idim))
  end do
end do

! renormalize
cc = TOFLOAT(X(mf_dotp)(mesh, st%d%dim, psii, psii))

call lalg_scal(mesh%np, st%d%dim, M_ONE/sqrt(cc), psii)

call states_elec_set_state(st, mesh, ist, ik, psii)
```

Example 2: Laplacian of a Gaussian

Let’s create a Gaussian centered on the origin

```
do ip = 1, this%mesh%np
   ff(ip) = bb*exp(-aa*sum(this%mesh%x(ip, :)**2)) + cc
 end do
```

Computing the Laplacian is done simply by calling

```
call dderivatives_perform(der%lapl, der, ff,op_ff)
```

Different derivative routines (gradient, Laplacien, divergence, curl, partial) are defined in *src/grid/derivatives.F90.*

The array ff is of size np_part, as we need to perform derivatives
Ghost points and boundary points are automatically set.

It is possible to ask for not setting them. This must be done with great care !

Example 3: Solving a Poisson equation

If we want to compute a Poisson equation

call dpoisson_solve(psolver, pot, dens)

Internally takes care of doing many operations, GPU transferts, MPI distribution, mesh to cube, cube to mesh,....

One should never call FFTs directly!

Quick summary

- To perform local operations: l*alg_basic_m/lalg_adv_m* modules (BLAS/Lapack)
- To perform derivatives: *derivatives_m*
- To compute global quantities: *mesh_function_m*
- To solver a Poisson equation: *poisson_solve*
- *To get/set states: states_elec_XX_state and batch_XX_state routines*

Problem with the previous approach

- XX_get_state routines imply copies and transfer of memory: very expensive !
- The same for set_state calls

- We want to remove these copies

- Does not work on GPUs

- Idea: manipulate the information directly where it is stored

Outline

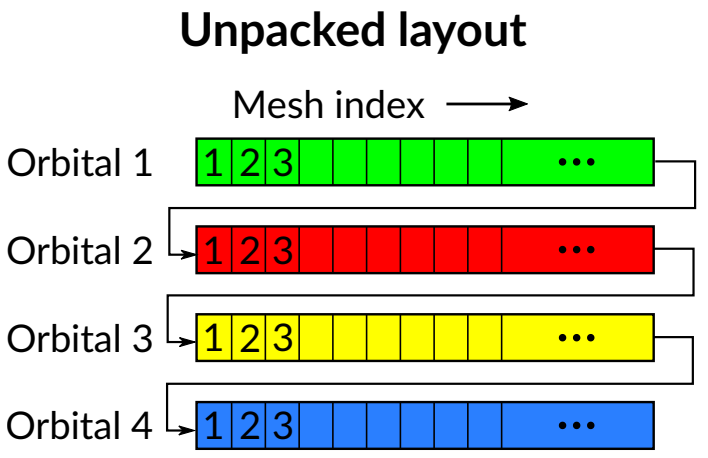
- Optimization strategy, profiling
- Techniques for efficient programming
- Parallelization
- Mesh functions
- **Batches**

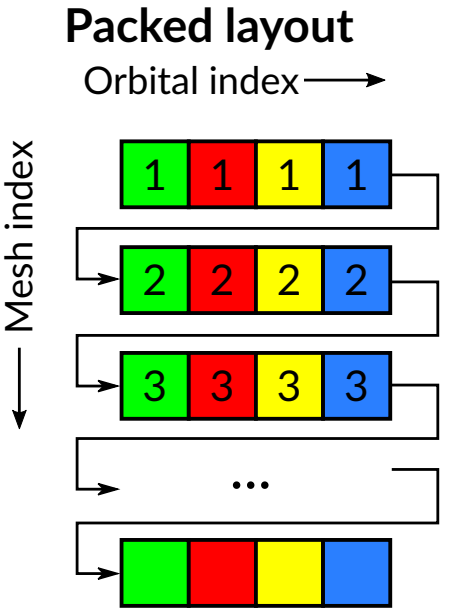
Higher level: batches

- Collection of mesh functions packed together
- Operations on batches are implemented on CPU and on GPU
- Using batches avoids transfers to/from the GPU
- Preferred way of manipulating wavefunctions

Data layout II: batches

- Aggregate several orbitals into one batch
- Operations done over batches
- 2 layouts:
  - Unpacked
  - Packed → vectorization, GPUs





Batch handling

- Batch can have 3 states:

CPU unpacked

CPU packed

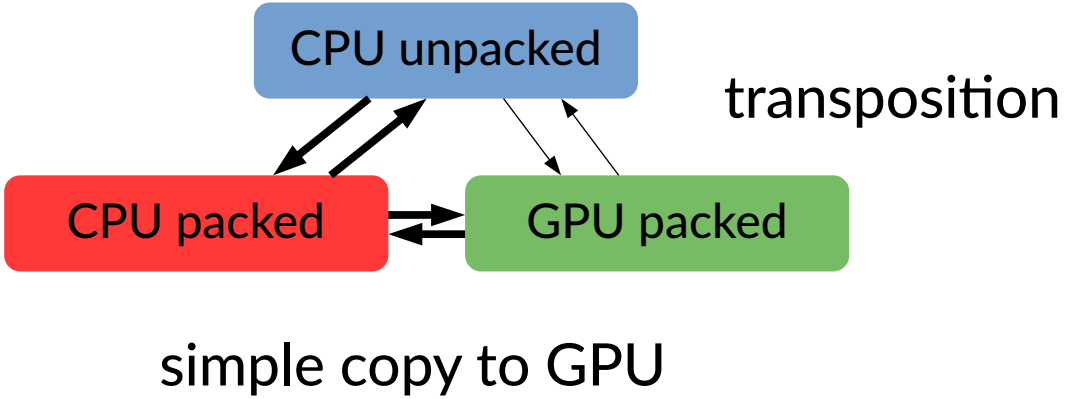
GPU packed

Batch handling

- Batch can have 3 states:

CPU unpacked CPU packed GPU packed

- Transitions

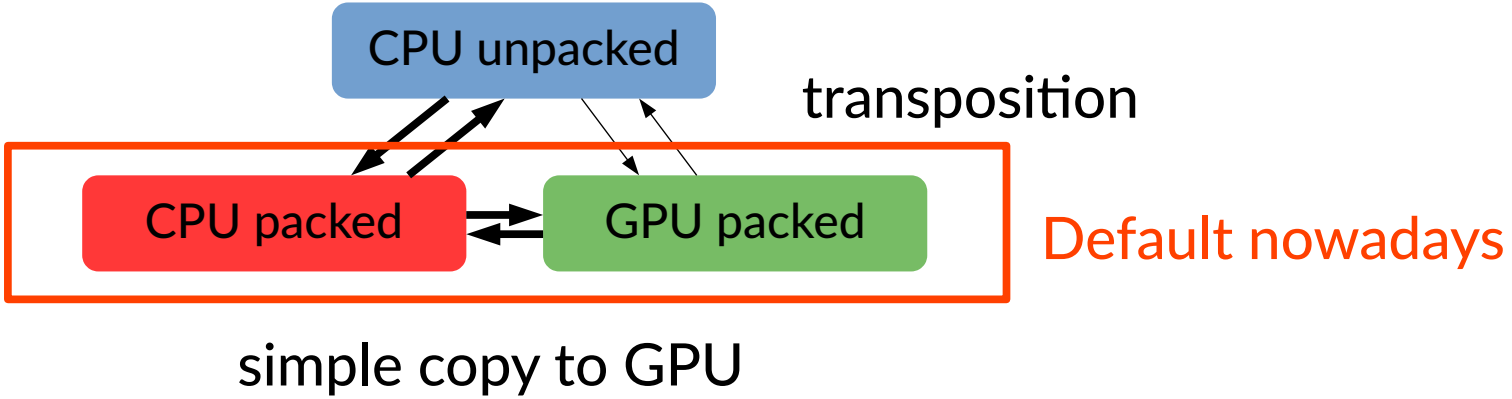


Batch handling

- Batch can have 3 states:

CPU unpacked CPU packed GPU packed

- Transitions



How to manipulate batches?

- Not directly (most of the time)
- Octopus provides dedicated routines
  - *batch_ops_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_m*: global operations like dot products

    Batch equivalent of *mesh_function_m* routines
  - *derivatives_m*: batch versions of the derivative routines

Example: Gram-Schmidt orthonormalization with batches

Adapted from mesh_batch_inc.F90.

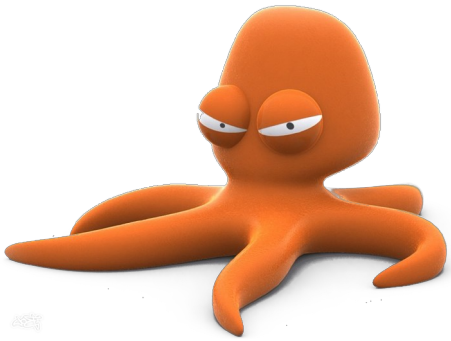
Orthonormalizes phib (mesh function) agains all the states in the array of batches psib(:)

```
do ist = 1, nst
  call X(mesh_batch_dotp_vector)(mesh, psib(ist), phib, ss(1:phib%nst,ist), reduce = .false.)
end do

if (mesh%parallel_in_domains) call mesh%allreduce(ss, dim = (/phib%nst, nst/))

do ist = 1, nst
  call batch_axpy(mesh%np, -ss(1:phib%nst,ist), psib(ist), phib, a_full = .false.)
end do

call X(mesh_batch_dotp_vector)(mesh, phib, phib, nrm2)
call batch_scal(mesh%np, M_ONE/sqrt(TOFLOAT(nrm2)), phib, a_full =.false.)
```

No get_state/set_state routine. All the data are manipulated in-place.

Summary

- Profiling: understand & optimize code
- Program with parallelization in mind
- Preferred usages:
  - Batches + operations
  - Mesh functions + operations



Tutorials

1) Profiling
2) Profiling on GPUs