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
<table>
<thead>
<tr>
<th>TAG</th>
<th>NUM CALLS</th>
<th>TOTAL TIME</th>
<th>TIME PER CALL</th>
<th>MFLOPS</th>
<th>MBYTES/S</th>
<th>%TIME</th>
<th>TOTAL TIME</th>
<th>TIME PER CALL</th>
<th>MFLOPS</th>
<th>MBYTES/S</th>
<th>%TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>zNL OPERATOR BATCH</td>
<td>4060</td>
<td>3.258858</td>
<td>0.000803</td>
<td>0.000209</td>
<td>2591.9</td>
<td>0.0</td>
<td>3.258858</td>
<td>0.000803</td>
<td>2591.9</td>
<td>0.0</td>
<td>35.3</td>
</tr>
<tr>
<td>PS FILTER</td>
<td>4</td>
<td>1.899388</td>
<td>0.047234</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.899388</td>
<td>0.047234</td>
<td>0.0</td>
<td>1.899388</td>
<td>0.0</td>
</tr>
<tr>
<td>SC PCONV</td>
<td>101</td>
<td>0.954465</td>
<td>0.009450</td>
<td>0.000319</td>
<td>0.0</td>
<td>0.0</td>
<td>0.954465</td>
<td>0.009450</td>
<td>0.0</td>
<td>0.954465</td>
<td>0.0</td>
</tr>
<tr>
<td>zvLPSI</td>
<td>2030</td>
<td>0.619451</td>
<td>0.000305</td>
<td>0.000166</td>
<td>563.1</td>
<td>2.2</td>
<td>0.619451</td>
<td>0.000305</td>
<td>563.1</td>
<td>0.0</td>
<td>6.7</td>
</tr>
<tr>
<td>zSET BC</td>
<td>2030</td>
<td>0.365798</td>
<td>0.000180</td>
<td>0.000098</td>
<td>0.0</td>
<td>0.0</td>
<td>0.365798</td>
<td>0.000180</td>
<td>0.0</td>
<td>0.365798</td>
<td>0.0</td>
</tr>
<tr>
<td>MESH INIT</td>
<td>3</td>
<td>0.357447</td>
<td>0.119149</td>
<td>0.000002</td>
<td>0.0</td>
<td>0.0</td>
<td>0.357447</td>
<td>0.119149</td>
<td>0.0</td>
<td>0.357447</td>
<td>0.0</td>
</tr>
<tr>
<td>zGHOST UPDATE START</td>
<td>2030</td>
<td>0.264602</td>
<td>0.001300</td>
<td>0.000069</td>
<td>0.0</td>
<td>0.0</td>
<td>0.264602</td>
<td>0.001300</td>
<td>0.0</td>
<td>0.264602</td>
<td>0.0</td>
</tr>
<tr>
<td>CALC DENSITY</td>
<td>595</td>
<td>0.222086</td>
<td>0.000440</td>
<td>0.000023</td>
<td>0.0</td>
<td>0.0</td>
<td>0.222086</td>
<td>0.000440</td>
<td>0.0</td>
<td>0.222086</td>
<td>0.0</td>
</tr>
<tr>
<td>ELECTRONS CONSTRUCTOR</td>
<td>1</td>
<td>0.383280</td>
<td>0.383280</td>
<td>0.383280</td>
<td>0.0</td>
<td>0.0</td>
<td>0.383280</td>
<td>0.383280</td>
<td>0.0</td>
<td>0.383280</td>
<td>0.0</td>
</tr>
<tr>
<td>zPROJ MAT SCATTER</td>
<td>12180</td>
<td>0.159651</td>
<td>0.000013</td>
<td>0.000002</td>
<td>73.0</td>
<td>0.0</td>
<td>0.159651</td>
<td>0.000013</td>
<td>73.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>COMPLETE RUN</td>
<td>1</td>
<td>9.226912</td>
<td>9.226912</td>
<td>9.226912</td>
<td>0.0</td>
<td>0.0</td>
<td>9.226912</td>
<td>9.226912</td>
<td>9.226912</td>
<td>0.0</td>
<td>100.0</td>
</tr>
<tr>
<td>HAMILTONIAN ELEC_INIT</td>
<td>1</td>
<td>2.000087</td>
<td>2.000087</td>
<td>2.000087</td>
<td>0.0</td>
<td>0.0</td>
<td>2.000087</td>
<td>2.000087</td>
<td>2.000087</td>
<td>0.0</td>
<td>101.6</td>
</tr>
<tr>
<td>zvNLPSI_MAT_BRA</td>
<td>2030</td>
<td>0.107823</td>
<td>0.000053</td>
<td>0.000034</td>
<td>1009.6</td>
<td>0.0</td>
<td>0.107823</td>
<td>0.000053</td>
<td>1009.6</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>LIBXC</td>
<td>202</td>
<td>0.073882</td>
<td>0.000366</td>
<td>0.000228</td>
<td>0.0</td>
<td>0.0</td>
<td>0.073882</td>
<td>0.000366</td>
<td>0.0</td>
<td>0.073882</td>
<td>0.0</td>
</tr>
<tr>
<td>zvNLPSI MAT REDUCE</td>
<td>2030</td>
<td>0.065831</td>
<td>0.000320</td>
<td>0.000191</td>
<td>0.0</td>
<td>0.0</td>
<td>0.065831</td>
<td>0.000320</td>
<td>0.0</td>
<td>0.065831</td>
<td>0.0</td>
</tr>
<tr>
<td>BLAS AXPY 4</td>
<td>3000</td>
<td>0.064741</td>
<td>0.000022</td>
<td>0.000008</td>
<td>5971.5</td>
<td>0.0</td>
<td>0.064741</td>
<td>0.000022</td>
<td>5971.5</td>
<td>0.0</td>
<td>0.7</td>
</tr>
<tr>
<td>zGHOST UPDATE WAIT</td>
<td>2030</td>
<td>0.043132</td>
<td>0.000021</td>
<td>0.000011</td>
<td>0.0</td>
<td>0.0</td>
<td>0.043132</td>
<td>0.000021</td>
<td>0.0</td>
<td>0.043132</td>
<td>0.0</td>
</tr>
<tr>
<td>BATCH COPY DATA TO</td>
<td>1750</td>
<td>0.042083</td>
<td>0.000240</td>
<td>0.000128</td>
<td>0.0</td>
<td>0.0</td>
<td>0.042083</td>
<td>0.000240</td>
<td>0.0</td>
<td>0.042083</td>
<td>0.0</td>
</tr>
<tr>
<td>zvNLPSI MAT KET</td>
<td>2030</td>
<td>0.248724</td>
<td>0.001230</td>
<td>0.000556</td>
<td>327.8</td>
<td>0.0</td>
<td>0.248724</td>
<td>0.001230</td>
<td>327.8</td>
<td>0.0</td>
<td>0.7</td>
</tr>
<tr>
<td>xcCUBE TO MESH</td>
<td>101</td>
<td>0.020717</td>
<td>0.000205</td>
<td>0.000191</td>
<td>0.0</td>
<td>0.0</td>
<td>0.020717</td>
<td>0.000205</td>
<td>0.0</td>
<td>0.020717</td>
<td>0.0</td>
</tr>
<tr>
<td>XC_LOCAL</td>
<td>101</td>
<td>0.093789</td>
<td>0.000929</td>
<td>0.000875</td>
<td>12.8</td>
<td>0.0</td>
<td>0.093789</td>
<td>0.000929</td>
<td>12.8</td>
<td>0.093789</td>
<td>0.0</td>
</tr>
<tr>
<td>zHAMILTONIAN</td>
<td>2030</td>
<td>4.943666</td>
<td>0.002435</td>
<td>0.000149</td>
<td>1817.6</td>
<td>0.0</td>
<td>4.943666</td>
<td>0.002435</td>
<td>1817.6</td>
<td>0.0</td>
<td>53.6</td>
</tr>
<tr>
<td>EXP TAYLOR BATCH</td>
<td>500</td>
<td>5.080677</td>
<td>0.010800</td>
<td>0.000267</td>
<td>1912.0</td>
<td>0.0</td>
<td>5.080677</td>
<td>0.010800</td>
<td>1912.0</td>
<td>0.0</td>
<td>54.2</td>
</tr>
<tr>
<td>dMESH TO CUBE</td>
<td>101</td>
<td>0.024494</td>
<td>0.000243</td>
<td>0.000231</td>
<td>0.0</td>
<td>0.0</td>
<td>0.024494</td>
<td>0.000243</td>
<td>0.0</td>
<td>0.024494</td>
<td>0.0</td>
</tr>
<tr>
<td>POISSON SOLVE</td>
<td>101</td>
<td>1.017069</td>
<td>0.010070</td>
<td>0.000940</td>
<td>0.0</td>
<td>0.0</td>
<td>1.017069</td>
<td>0.010070</td>
<td>0.0</td>
<td>1.017069</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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:
  
  ```bash
  ./configure --enable-cuda --enable-nvtx ...
  ```

- Enable profiling (ProfilingMode = prof_time)

- Install Nsight systems (or use nsight_systems/2021 module on MPCDF systems)

- Run Nsight:
  
  ```bash
  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
\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

Column-major order
\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

Image source: Wikipedia, Author: Cmglee
Example code: phase

- **First dimension: \texttt{ii}**
  - Fastest index
  - Inner loop
- **Second dimension: \texttt{ip}**
  - Slowest index
  - Outer loop

```fortran
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

<table>
<thead>
<tr>
<th>MPI</th>
<th>K points / Spin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kohn Sham states</td>
</tr>
<tr>
<td></td>
<td>Real space domains</td>
</tr>
<tr>
<td>OpenMP</td>
<td>CUDA</td>
</tr>
<tr>
<td>Vectorization</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>GPU</td>
</tr>
</tbody>
</table>

- Vectorization
- Kohn Sham states
- Real space domains
- OpenMP
- CUDA
- K points / Spin
- CPU
- GPU
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

```fortran
  do iqn = st%d%d%kpt%start, st%d%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 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
What we **should not do**!

must

Taken from *src/hamiltonian/kb_projector_inc.F90*

```fortran
  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!

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 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 $r_i$: $\text{array}(i)=f(r_i)$
- 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
Data layout

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

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 $r_i$:
  $$\text{array}(i) = f(r_i)$$

- 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

Needed for derivatives
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:

```fortran
    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**: 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):

```plaintext
    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 *psibnst*.

*np*: number of points requested. Usually *meshnp*, sometimes *meshnp_part*.

*psi*(1:*meshnp*, 1:*stdim*): a wavefunction (or Pauli spinor)

The batch carries 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):

\[ u_1 = v_1, \]
\[ u_2 = v_2 - \text{proj}_{u_1}(v_2), \]
\[ u_3 = v_3 - \text{proj}_{u_1}(v_3) - \text{proj}_{u_2}(v_3), \]
\[ u_4 = v_4 - \text{proj}_{u_1}(v_4) - \text{proj}_{u_2}(v_4) - \text{proj}_{u_3}(v_4), \]
\[ \vdots \]
\[ u_k = v_k - \sum_{j=1}^{k-1} \text{proj}_{u_j}(v_k), \]

\[ e_1 = \frac{u_1}{||u_1||}, \]
\[ e_2 = \frac{u_2}{||u_2||}, \]
\[ e_3 = \frac{u_3}{||u_3||}, \]
\[ e_4 = \frac{u_4}{||u_4||}, \]
\[ \vdots \]
\[ e_k = \frac{u_k}{||u_k||}. \]

Where

\[ \text{proj}_u(v) = \frac{\langle u, v \rangle}{\langle u, u \rangle} u, \]

Basic operations: dot product, norm, \( y = a^*x + y \)
Example 1: Gram-Schimdt orthonormalization

Taken from src/states/states_elec_calc_inc.F90

```fortran
! 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

\[
\text{do } ip = 1, \text{this}\%\text{mesh}\%\text{np} \\
\quad \text{ff}(ip) = bb \times \exp(-aa \times \text{sum(this}\%\text{mesh}\%x(ip, :)**2)) + cc \\
\text{end do}
\]

Computing the Laplacian is done simply by calling

\[
\text{call dderivatives\_perform(der\%lapl, der, ff, op\_ff)}
\]

Different derivative routines (gradient, Laplacian, 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(p_solver, 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: `lalg_basic_m/lalg_adv_m` modules (BLAS/Lapack)
- To perform derivatives: `derivatives_m`
- To compute global quantities: `mesh_function_m`
- To solve 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

  - simple copy to GPU
  - transposition
Batch handling

• Batch can have 3 states:
  - CPU unpacked
  - CPU packed
  - GPU packed

• Transitions
  - Simple copy to GPU
  - Transposition
  - Default nowadays

  simple copy to GPU

transposition
How to manipulate batches?

• Not directly (most of the time)
• Octopus provides dedicated routines
  – \textit{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,...)
  – \textit{mesh\_batch\_m}: global operations like dot products
    Batch equivalent of \textit{mesh\_function\_m} routines
  – \textit{derivatives\_m}: batch versions of the derivative routines
Example: Gram-Schmidt orthonormalization with batches

Adapted from mesh_batch_inc.F90.

Orthonormalizes \text{phib} (mesh function) against all the states in the array of batches \text{psib}(::)

\begin{verbatim}
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.)
\end{verbatim}

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