# Octopus: High Performance Computing

Martin Lüders and the Octopus developers

Octopus Advanced Course 2023, MPSD Hamburg

# Introduction

Up to now: Octopus on your laptop.

# Introduction

Faster results needed?
Go parallel!


©MPCDF

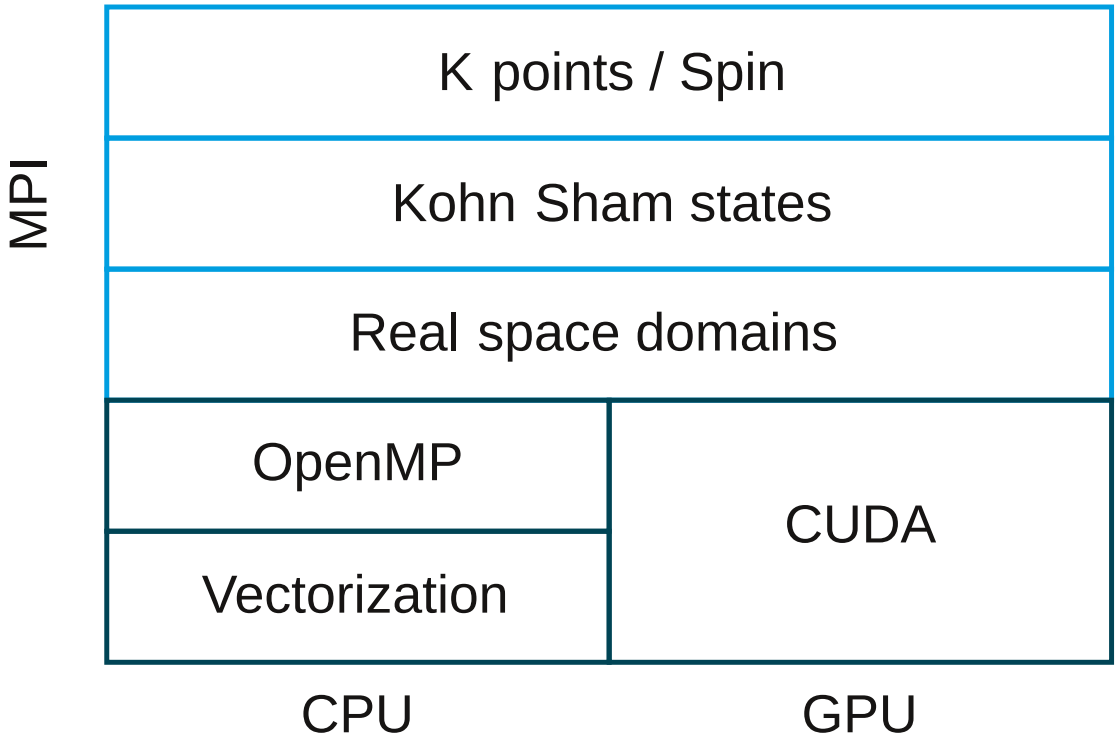
# High Performance Computing

- Todays smartphones faster than HPC in the 1980s.
- For many years: exponential increase in clock speed
- Now: clock speed saturated, move to increased parallelism
- Adapting to new hardware is much more complicated
- Need to combine different technologies:
  MPI, OpenMP, GPU, Vectorization

# Levels of parallelism

- Hierarchy in HPC systems:
  - Cluster: Many compute nodes: MPI
  - Node: several sockets with CPUs, maybe some GPUs
  - CPU: several cores: MPI, OpenMP
  - GPU: many cores: OpenCL, CUDA, etc.
  - Core: vectorization, pipelining: SIMD
- Best performance: exploit all levels

# Parallelization strategies in Octopus



# Parallelization in k-points

- Different k points are mostly independent
- Each processor group handles one or several k points
- Weakest coupling

# Parallelization in states

- Each processor handles a group of states
- Efficient for time propagation
- Also used for ground state, but stronger coupling (orthogonalization, subspace diagonalization)

# Parallelization in domains

- Each processor handles a region in space
- Need to communicate ghost points
- Integrals require reduction over all regions
- Least efficient parallelization strategy
- Watch ratio of inner points to ghost points!

# OpenMP parallelization

- Shared-memory approach: threads access the same memory
- Octopus: loops over grid can use OpenMP
- No ghost points needed
- Similar to domain parallelization
- Number of local points needs to be large enough
- Can be efficient using up to 12 threads
- OpenMP threads should be on the same socket

# Vectorization

- Modern CPUs: several floating point operations in one instruction
- Needed to exploit full performance
- In Octopus:
  - Data structures designed to facilitate vectorization
  - Hand-crafted kernels for stencil operation

# Controlling the parallelization

- Input options:
  - ParKPoints
  - ParStates
  - ParDomains
  - ParOther (e.g. for Casida)
- control number of processors for each strategy
- can also be
  - auto
  - no
- Default:
  - TD: auto for all
  - GS: auto for all except ParStates

# Choosing the number of processors

- Automatic setting is not always the best
- Product of processors per strategy = total number of processors
- If OpenMP is used: product of OMP threads and MPI tasks = total number of processors

# MPI framework

Several layers of abstraction:

- `mpi.F90`
  - `mpi_grp_t`
  - mpi routines for the group
- `comm.F90`
  - `comm_allreduce(grp, aa)`: interface to reductions of various types
- `multicomm.F90`
  - construct communicators for various strategies
- `mpi_lib.F90`
  - various other MPI routines

In general, if you need more than a simple reduction, consult the core developers, as good (or bad) MPI strategies can have impact on the code performance.

# k-point parallelization

- k-points are mostly independent.
- Often only reduction required. (Brillouin zone summation)
  - Process local k-points: `ik` from `st%d%kpt%start` to `st%d%kpt%end`
  - reduction over mpi groups
  - example: eigenvalue sum

```
tot = M_ZERO
do ik = st%d%kpt%start, st%d%kpt%end
  tot = tot + st%d%kweights(ik) * sum(st%occ(st%st_start:st%st_end, ik) * &
    st%eigenval(st%st_start:st%st_end, ik))
end do

if (st%parallel_in_states .or. st%d%kpt%parallel) &
  call comm_allreduce(st%st_kpt_mpi_grp, tot)
```

# State parallelization

- TD-propagation: states are mostly independent.
- Only reduction required for total density, energy, etc.
- Same method as for k-points.

- Each processor owns a subset of states (`group%block_start:group%block_end`)
- get node containing a block: `group%block_node(iblock)`
- get a copy of a remote block: `states_elec_parallel_get_block()`
- more utilities in `states_elec_parallel.F90`

# Parallel domain decomposition

- Mostly performed under the hood:
- ghost exchange is performed unless specified otherwise (optional flag)
- in some cases, it is better to postpone the reduction, see e.g. `X(calc_expectation_value)`

# Example: X(calculate_expectation_values)

```
...
R_TYPE,                    intent(out)    :: eigen(st%st_start:, st%d%kpt%start:)
...

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))
```

## Reduction over states and k-points performed in states_elec_eigenvalue_sum()

```
if (st%parallel_in_states .or. st%d%kpt%parallel) &
  call comm_allreduce(st%st_kpt_mpi_grp, tot)
```

# OpenMP

- Shared memory parallelization
  - No communication necessary
  - Only possible within one node
  - Take care of private variables, and race conditions!
- Independent of MPI parallelization scheme
- Perform expensive loops with OpenMP threading, e.g loops over mesh points

```
!$omp parallel do simd schedule(static) private(ip)
do pos = 1, mesh%np
  ip = mesh_global2local(mesh, recv_indices(pos))
  ASSERT(ip /= 0)
  do ist = 1, nstl
    aa%X(ff_pack)(ist, ip) = recv_buffer(ist, pos)
  end do
end do
```

# GPU arcitecture

- Many Streaming Multiprocessors
- Limited flexibility
- in general GPUs have their own memory
- High latency, high throughput
- architecture keeps changing (e.g. direct NVLink, etc.)

# GPU programming models

Different strategies and languag extensions to program GPUs

- OpenACC: preprocerssor directives
- OpenMP: preprocerssor directives
- OpenCL: GPU kernels
- CUDA: GPU kernels
- HIP: GPU kernels

Octopus started with OpenCL and then introduced a CUDA compatibility layer.

# GPU kernels

Programming model in CUDA and OpenCL:

- GPU kernels: small routines, which execute one element of a loop
- GPU kernels live in `share/opencl`
- kernels are usually very short routines
- All data needs to be transferred to the GPU and back
- try to keep data on GPU as long as possible
- try to overlap communication and computation

# GPU in Octopus

- We try to encapsulate GPU code
- Many batch functions work automatically on GPU
- Batches have three states:
  - `UNPACKED`: CPU, normal storage order (grid index first)
  - `PACKED`: CPU, transposed order (state index first)
  - `DEVICE_PACKED`: GPU, transposed order (state index first)
- In most cases, you won't need to touch any GPU code.

- Use batch functions whenever you can!