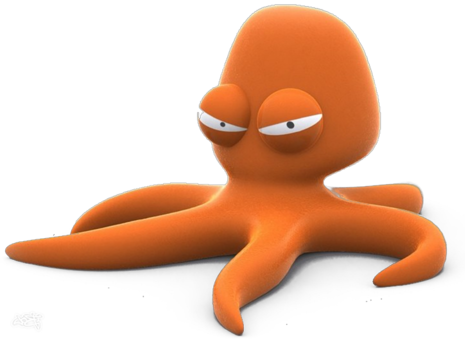
MAX PLANCK
COMPUTING & DATA FACILITY

MAX PLANCK INSTITUTE
FOR THE STRUCTURE AND DYNAMICS OF MATTER

Octopus on HPC systems: Parallelization and GPUs

Sebastian Ohlmann
Max Planck Computing and Data Facility, Garching

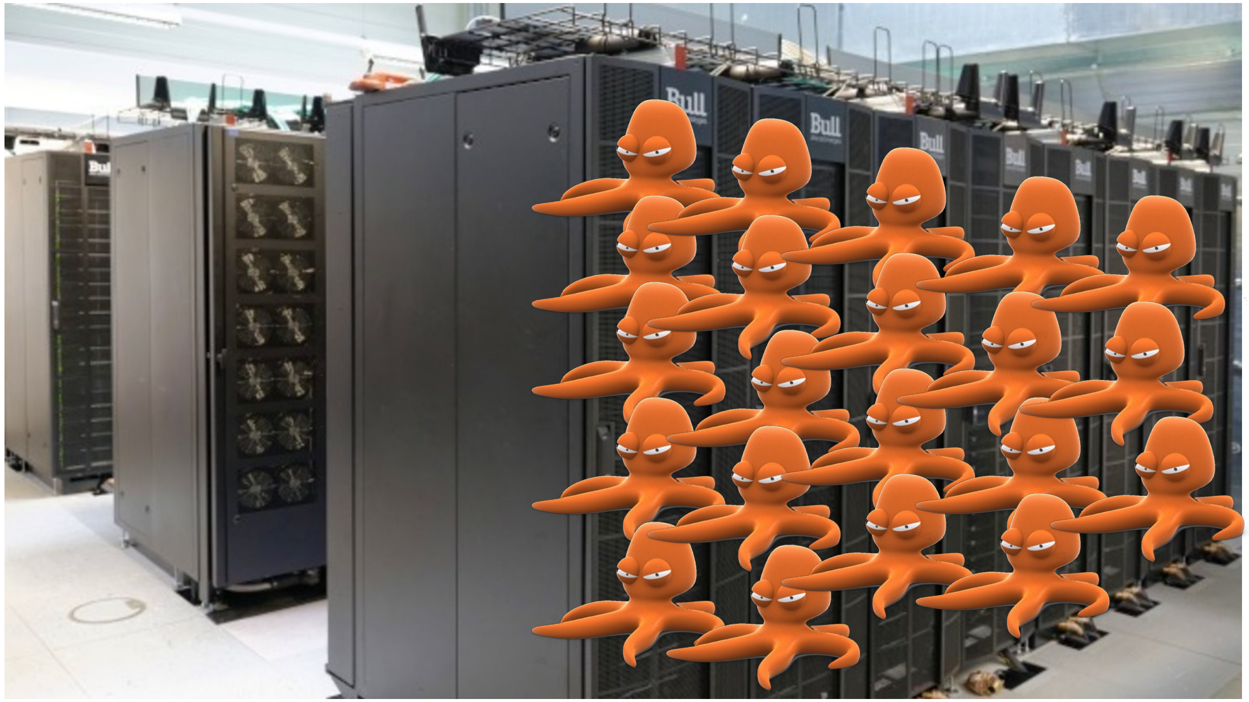
Octopus basics course, 7.9.2021



Up to now:
Octopus on your laptop

Faster results needed?

Go parallel!



©MPCDF

Outline

- High performance computing
- Parallelization strategies in octopus
- Guidelines for efficient usage
- Using GPUs with octopus
- Tutorials

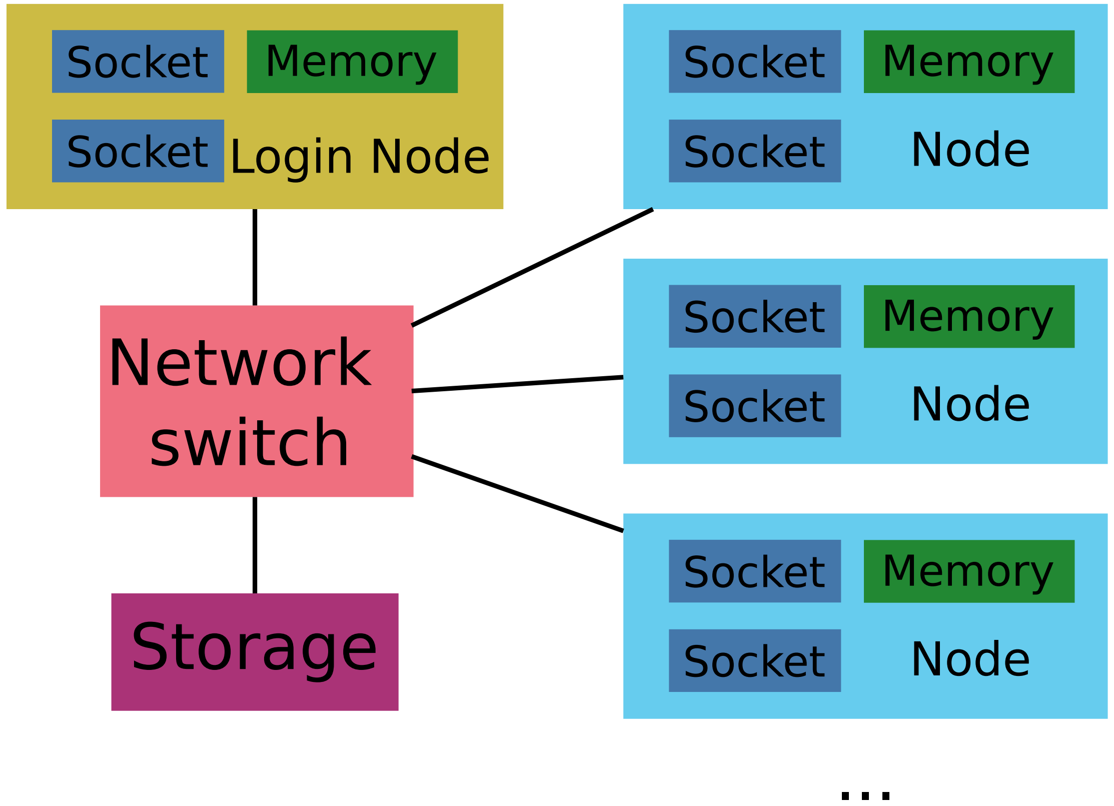
High performance computing

- Also HPC = supercomputing
- Definition difficult: today’s smartphones better than supercomputers from 50 years ago
- Utilize hardware to the fullest
- Parallel computing important
  → distribute computations to get faster results

Levels of parallelism

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

Architecture of an HPC system



Resource management

- Goal: maximize resource utilization
- Users submit compute jobs to a queue
- Need to specify required resources
- Scheduler assigns jobs to resources
- Scheduler starts/ends jobs
- Widely used: **slurm** → learn more in tutorial

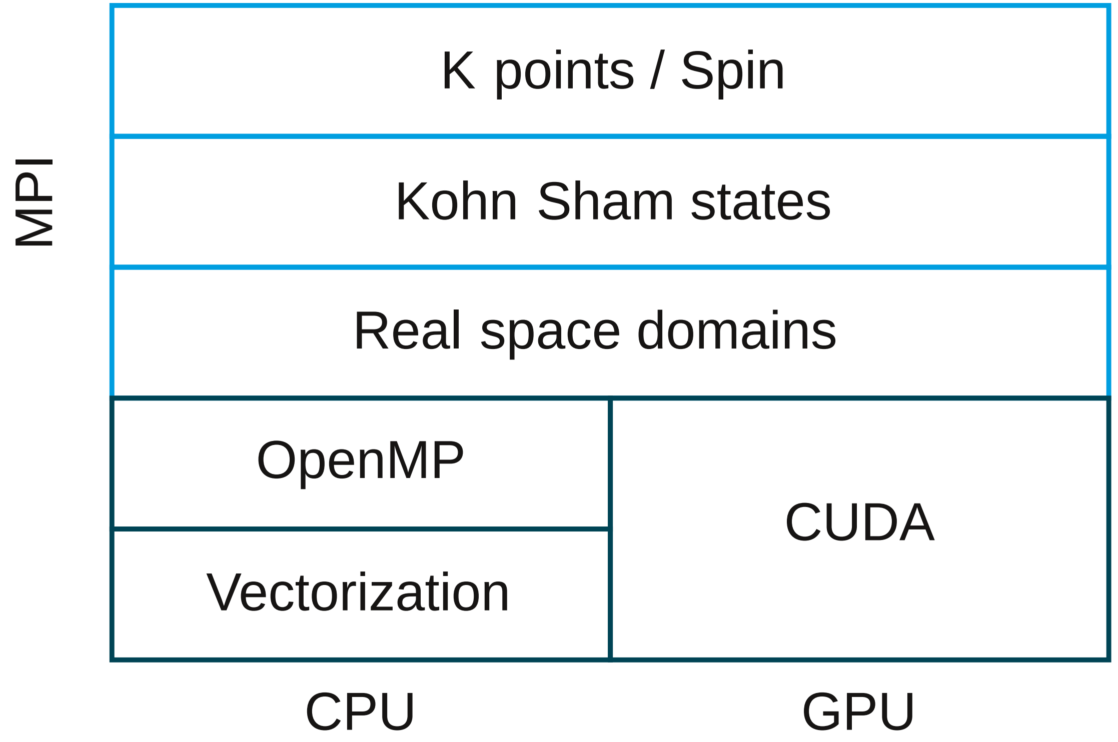
Parallelization in Octopus

- Central object: Kohn-Sham wavefunctions
- Several dimensions:
  - K points
  - State index
  - Real-space grid index
- Idea:
  - Distribute wavefunctions over all these dimensions
  - Every process works on local part of wavefunctions
  - Communication needed for synchronization

Parallelization approach

- Distributed-memory parallelization: **MPI**
  → scale to multiple compute nodes
- Shared-memory parallelization: **OpenMP**
  → inside one node
- Vectorization → inside CPU cores
- **GPUs** for offloading computations from CPU

Parallelization strategies



Parallelization in k points/spin

- Different k points independent
- Each process handles one or several k points
- Weakest coupling

Parallelization in states

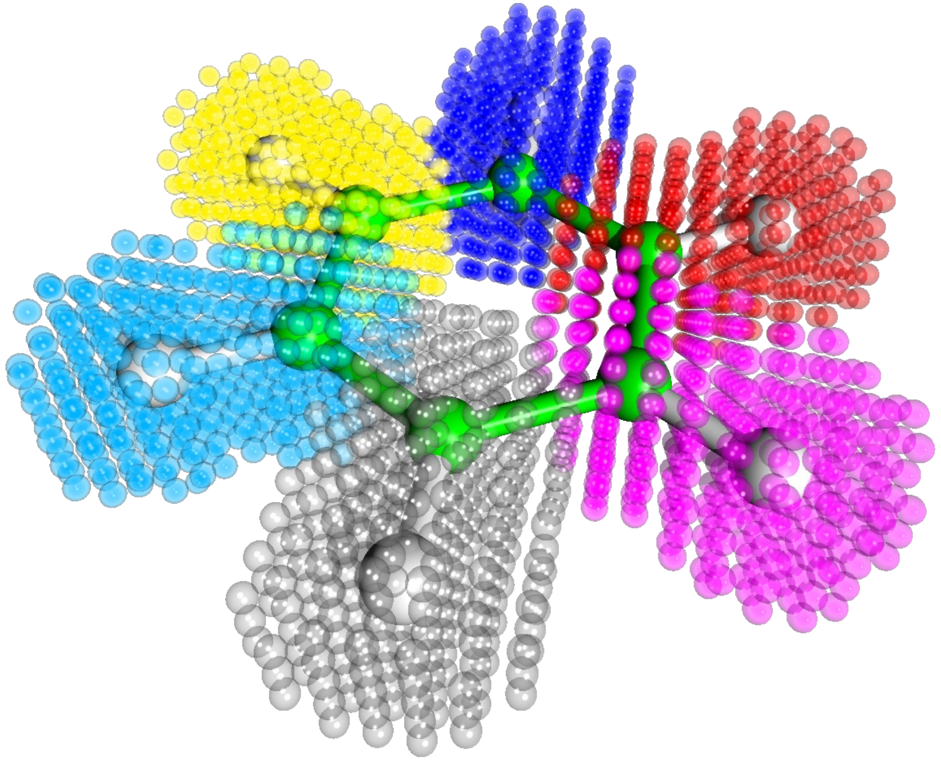
- Each process handles a group of states
- Efficient for time propagation
- Also used for ground state, but stronger coupling
  (orthogonalization, subspace diagonalization)

Parallelization in domains

- Each process handles points of a region in space
- Derivatives: finite differences using a stencil
- Information from neighbors needed → **ghost points**
- Integrals: performed locally and summed over all domains
- Introduces more communication & stronger coupling
- Less efficient than other strategies

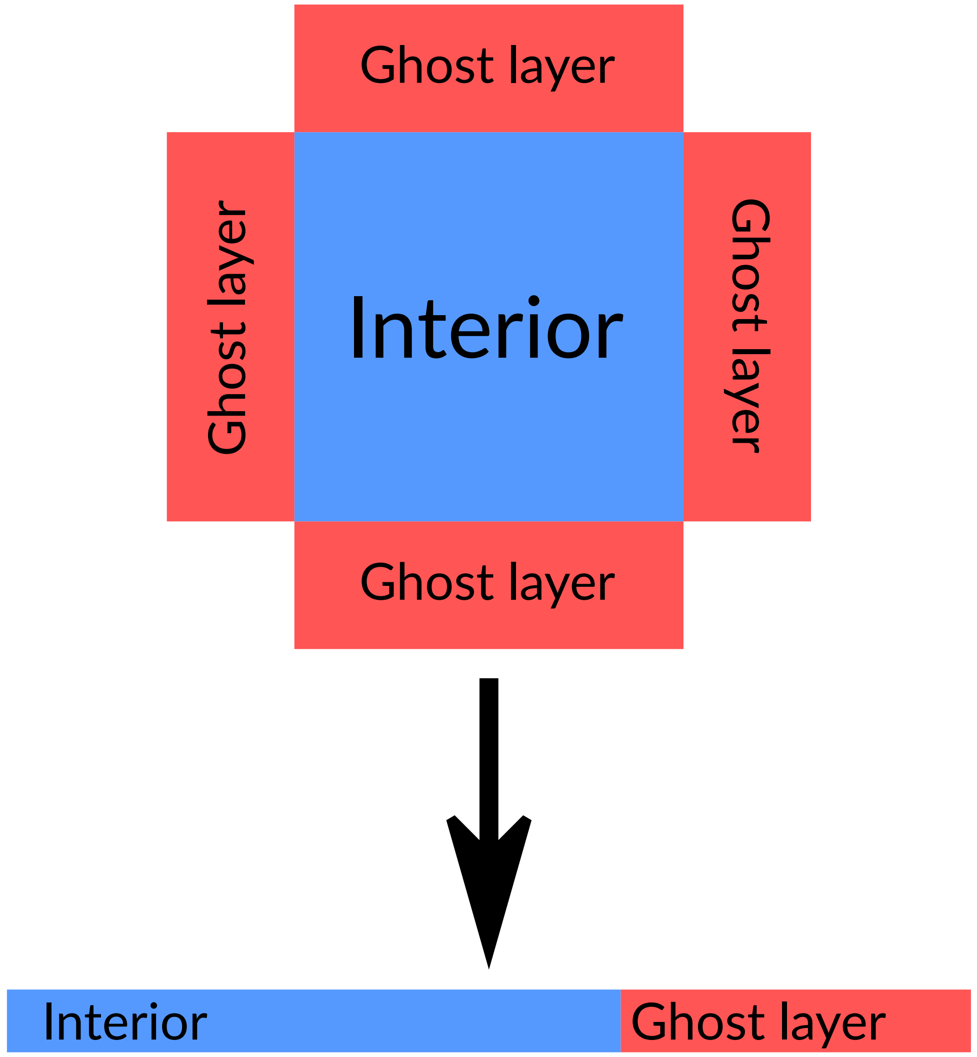
Partitioning

- Uses METIS library
- Minimize load imbalance and communication
  - Same number of points
  - Small boundary surfaces



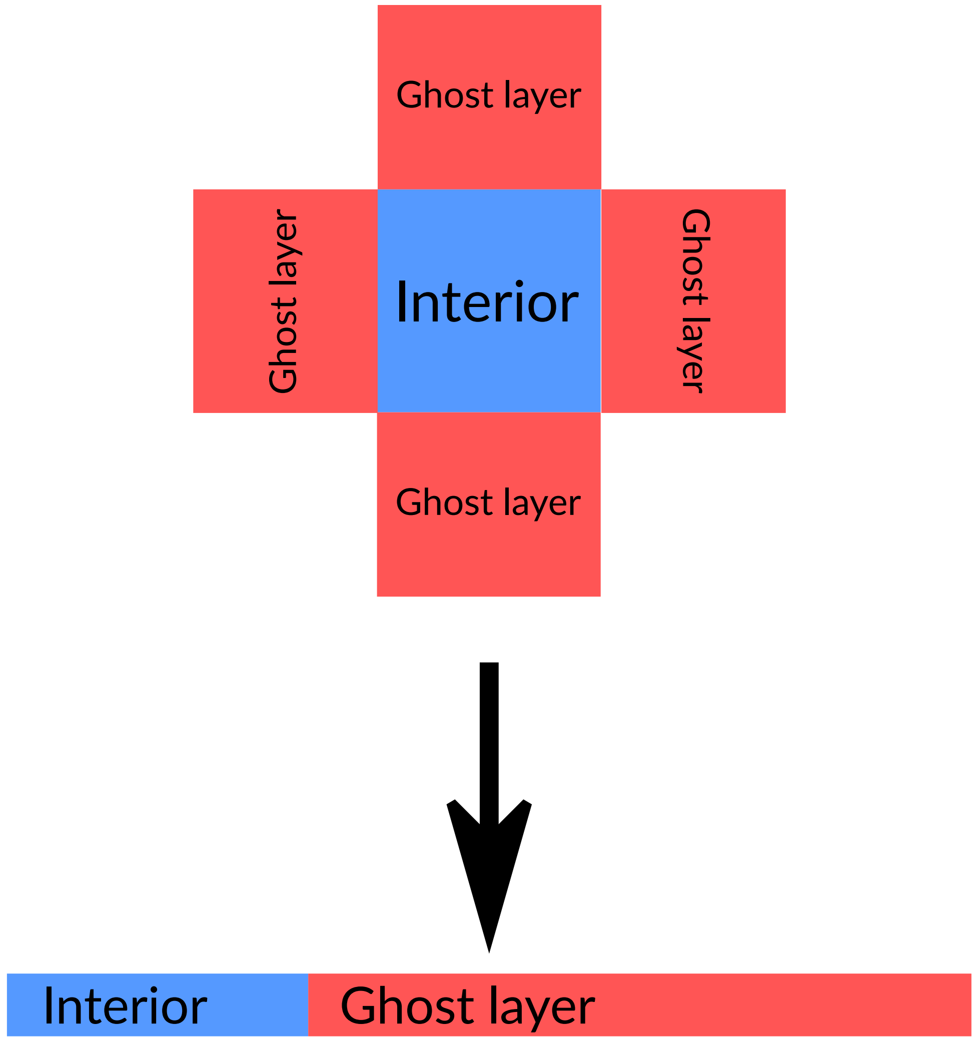
Domain parallelization

- Work on local points
- Ghost points: needed for stencil
- Communication: for updating them



Too many ghost points

- Large ratio of ghost to inner points (> 25%)
- Communication overhead too large
- Not enough local work
- **Inefficient!**
  → use less cores



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 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 number of processors

- Automatic setting not always best option
- Setting by hand often yields better results
- Product of processors in each direction = total number of processors
- If OpenMP used: product of processors x OpenMP threads = total number of processors

Parallelization example I

- Large molecule (finite system, no k points)
  - 268 states
  - 260000 grid points
- Run on cobra (40 cores per node)
- 1 node: 40 cores = 2^3 x 5
  - ParStates=40 → 7 or 6 states per process
  - ParStates=20, ParDomains=2 → 13 or 14 states per process, 130000 points per process
  - ParStates=20, OpenMP=2 (instead of ParDomains)
  - ParStates=10, ParDomains=2, OpenMP=2

Parallelization example II

- Small solid
  - 5x5x5 = 125 k points
  - 16 states
  - 8000 grid points (→ too small for parallelization)
- Run on cobra (40 cores per node)
- 1 node: 40 cores = 2^3 x 5
  - ParKPoints=10, ParStates=4 → 13 or 12 k points per process, 4 states per process
  - ParKPoints=20, ParStates=2 → 7 or 6 k points per process, 8 states per process
- Imbalance not always avoidable

How do I know if I run the code efficiently?

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 4
- Domains: ratio ghost/local points <25%

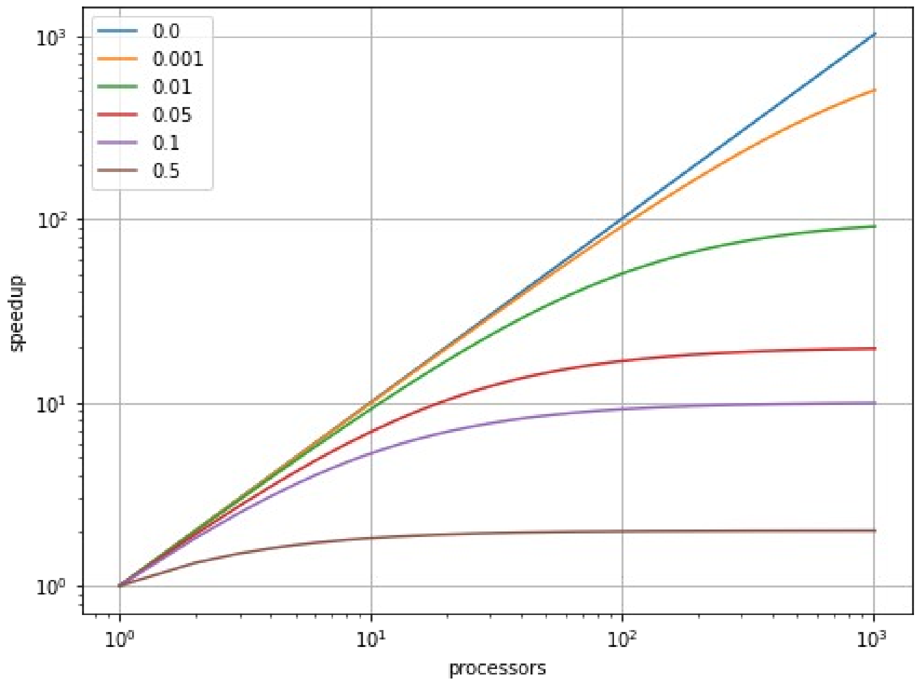
Scaling

- Expectation: using 2 processors instead of 1 → twice as fast
- In reality: not the case!
- Problems reducing efficiency:
  - Not all operations parallelized
  - Overhead of parallelization scheme (communication, bookkeeping, …)
- Analyze scaling to find efficient configuration

Terminology

- Time on N processors: T(N)
- Speed-up: S = T(1)/T(N)
- Ideal speed-up: S_{ideal} = N/1
- Parallel efficiency: $\epsilon = S/S_{ideal}$

Amdahl's law



- Speed-up S for serial fraction f on N processes:
  $S = 1/(f + (1-f)/N)$
- Upper limit: $1/f$
- For $f=10\% \rightarrow S \leq 10$!
- Gives upper limit on achievable speed-up

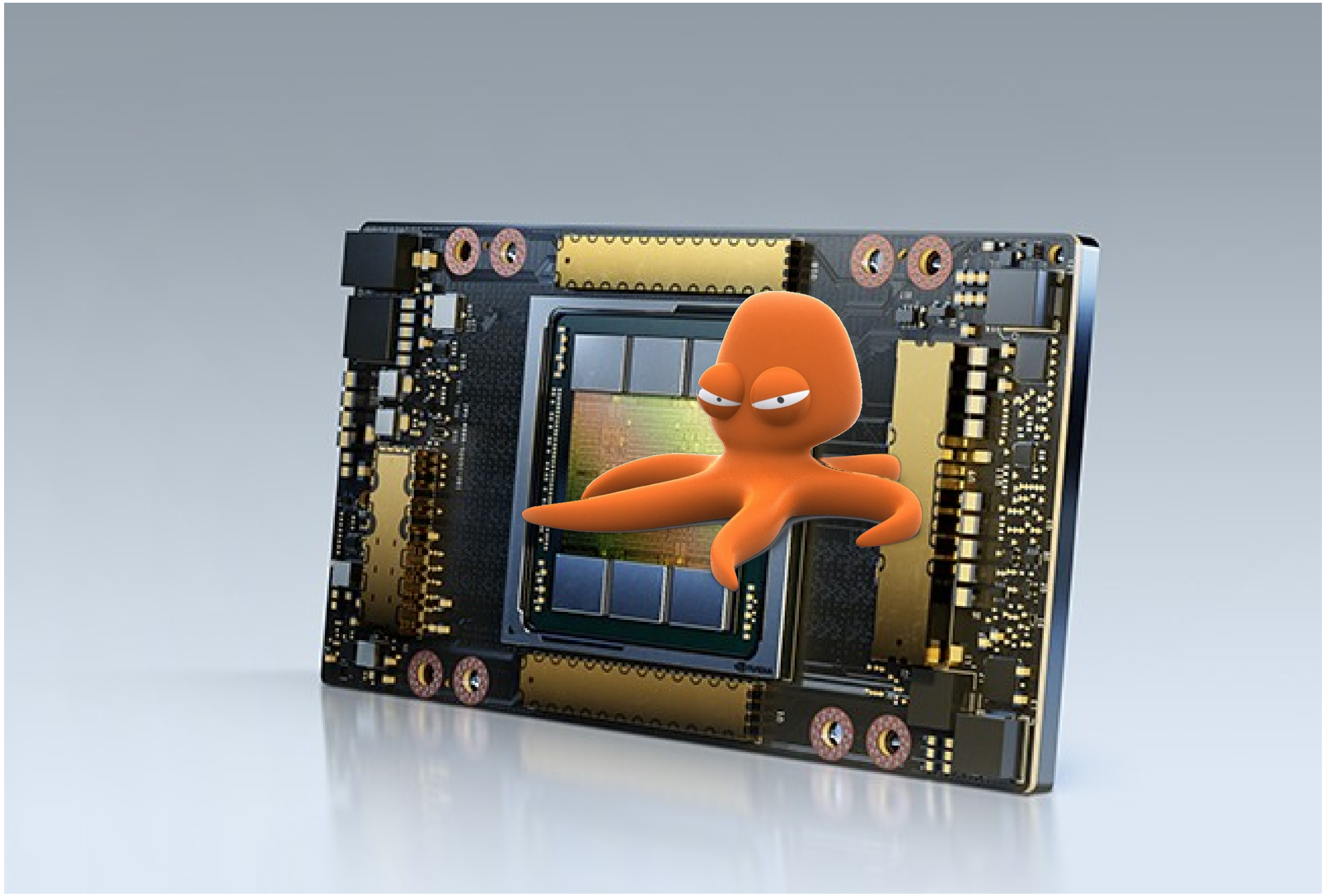
Scaling analysis

- Goal: determine by experiments up to which point octopus scales for certain input
- Strong scaling:
  - Run octopus for 1, 2, 4, 8, 16, … cores (or nodes)
  - Compute speed-up
  - Compare to ideal speed-up in scaling plot (speed-up vs. cores in log-log plot)
  - Problems where curve deviates from ideal scaling
  - Efficiency should be above 70%

Why should I care?

- HPC systems: large, but finite and **shared** resources
- *Efficient usage*: more simulations (and science) can be done in total by all members of the group
- *Inefficient usage*: less simulations can be done, longer waiting times for all members of the group

Octopus on GPUs



©Nvidia

Octopus on GPUs

- Implementation: uses CUDA
- Targets only NVIDIA GPUs at the moment
- Code needs to be compiled with CUDA support
- No special settings in input file needed
- Only efficient on HPC GPUs (need double precision operations!)

Features on GPUs

- Most efficient: time propagations for large systems
- Also working: ground state → use RMMDIIS eigensolver
- Some features do not work/are inefficient (e.g., spin-orbit coupling, DFT+U, hybrid functionals)

Guidelines for efficiency

- One process per GPU on each node
- Many states: min. 16-32 states per process
- Large grids: enough points needed to saturate GPUs
- Domain parallelization introduces communication overhead (GPU ↔ CPU)

How to get started

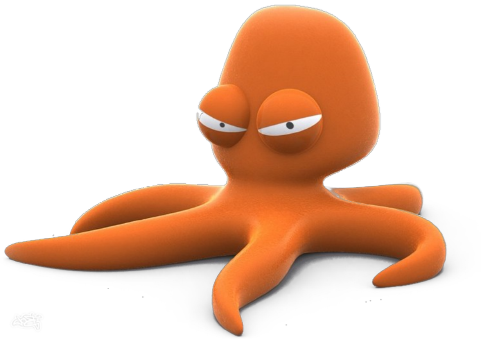
- Compile code with CUDA support
- Run on a system with NVIDIA GPUs
- At MPCDF: use octopus-gpu module
- Compare timings to CPU run
- In case of issues or inefficiencies, let the developers know!

Why use GPUs?

- For suitable setups, using GPUs can be 10 times faster than on CPUs (on same number of nodes)
  - Faster time to solution
  - More efficient
- Large GPU resources now and in future
- Try it out!

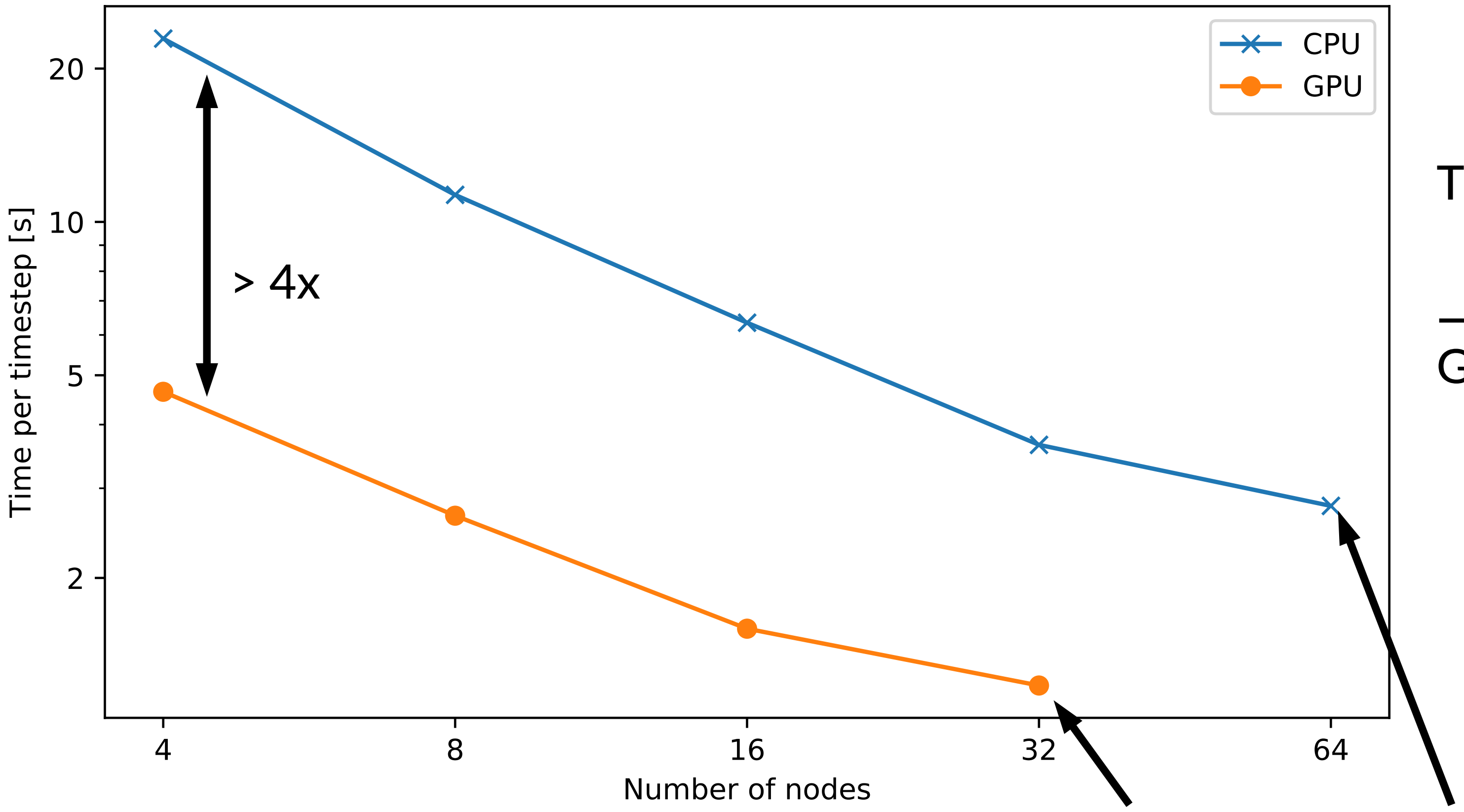
Tutorials

1) Using MPCDF systems
2) Slurm usage
3) Parallelization in octopus
4) Scaling
5) Octopus on GPUs



Backup slides

Comparison on cobra: CPU vs. GPU



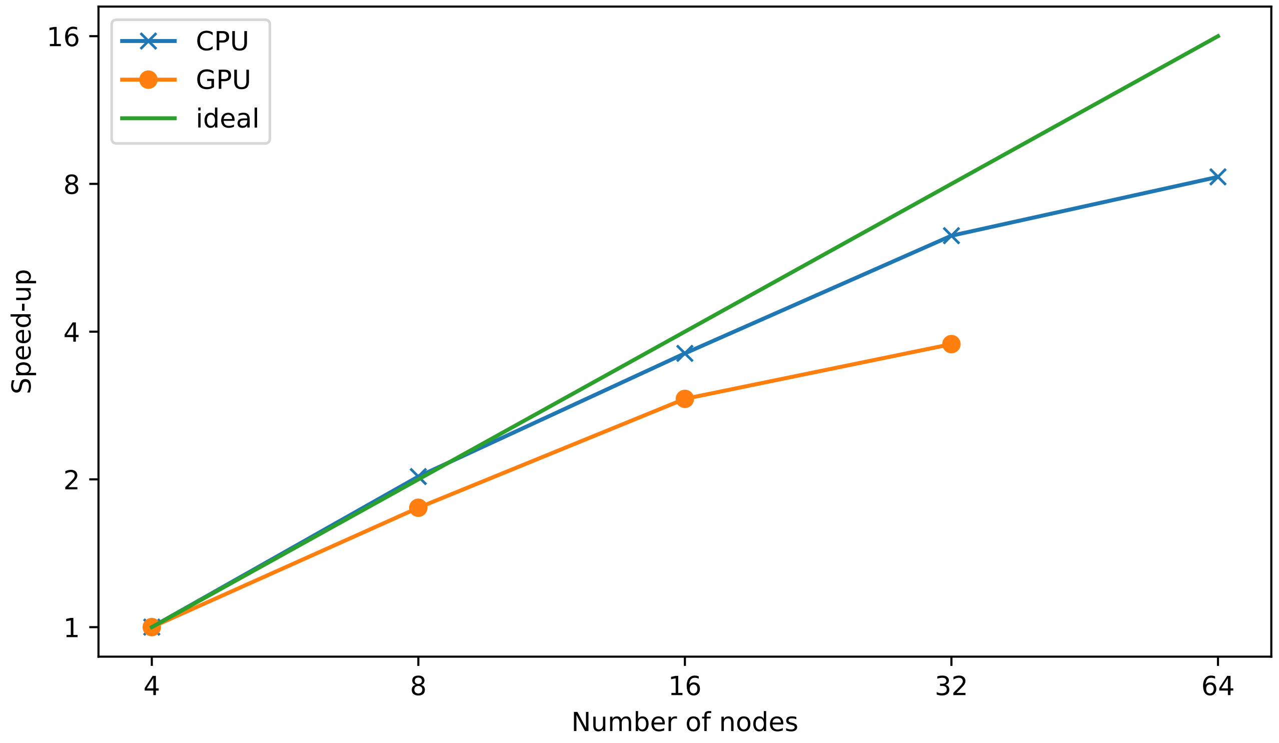
64 GPUs

2560 cores

TCO of cobra nodes:
GPU ~ 2.8 CPU

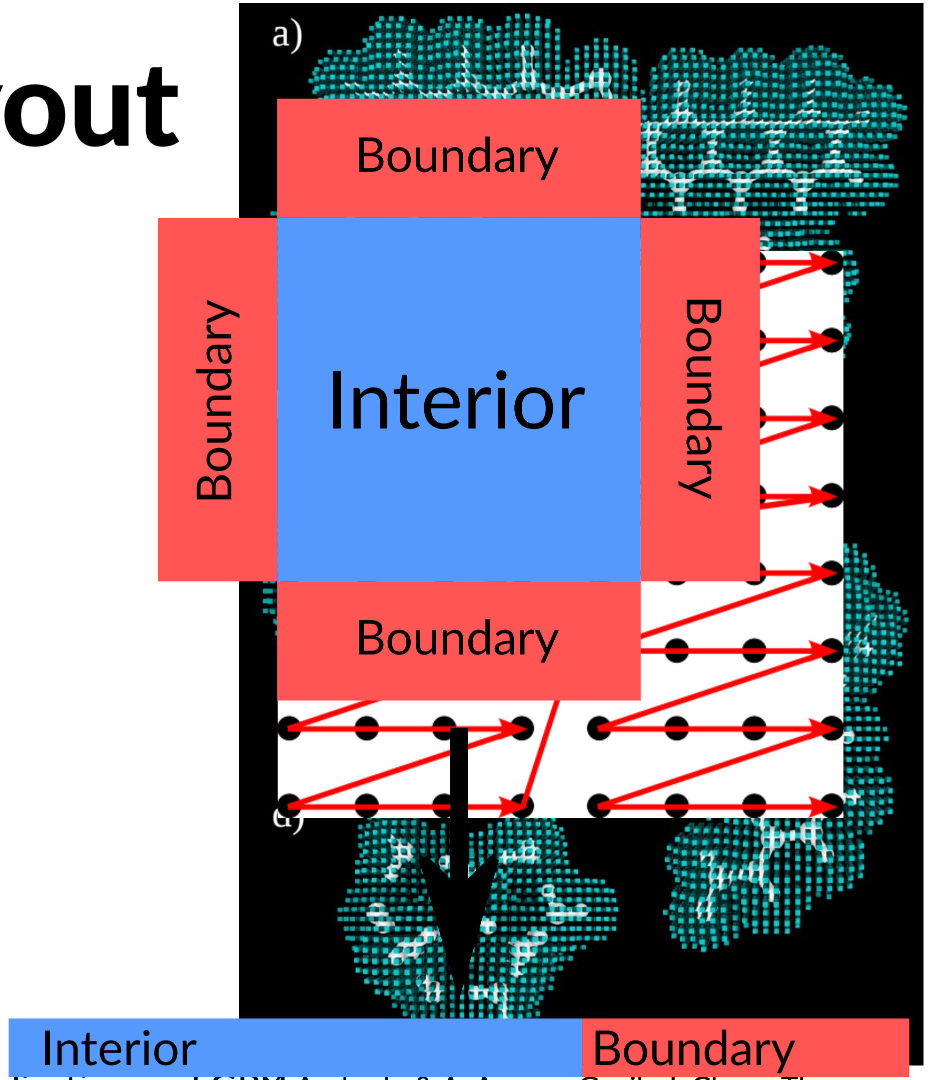
→ cost-efficient on GPUs!

Scaling on GPU nodes



Data layout

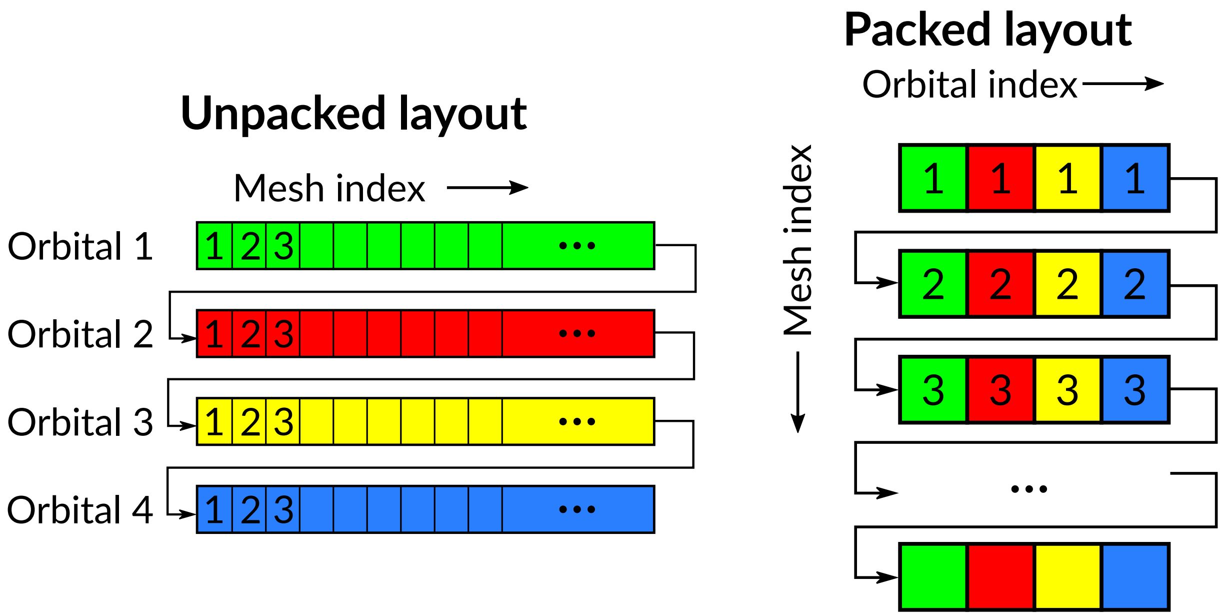
- Real-space grid for FD
- Complicated shape possible, e.g. molecules
- Cache-aware mapping to 1D array
- 1D data layout: 2 blocks
  - Interior points
  - Boundary/ghost points



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

Data layout II: batches

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

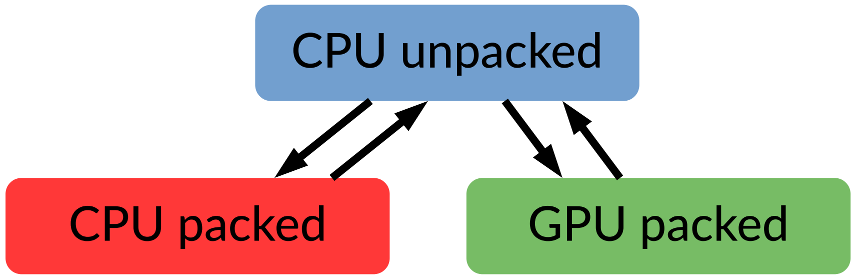


Batch handling

- Batch can have 3 states:

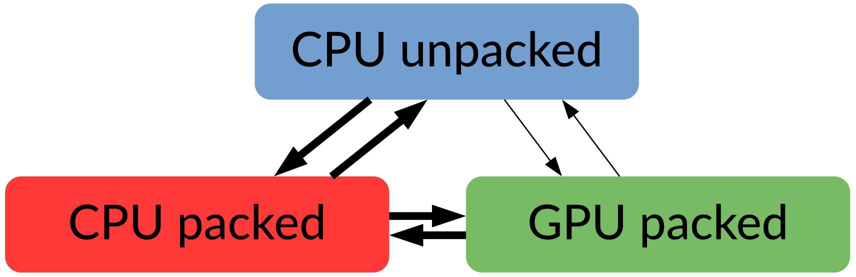
CPU unpacked　CPU packed　GPU packed

- Transitions before:



→ always involves transposition

- Transitions now:



→ simple copy to GPU