Octopus: High Performance Computing

Martin Lüders and the Octopus developers

Octopus Advanced Course 2023, MPSD Hamburg
Introduction

Up to now: Octopus on your laptop.
Faster results needed?
Go parallel!
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

MPI

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

CPU

GPU

CUDA
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
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

```fortran
  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: \texttt{X(calculate\_expectation\_values)}

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

```c
!$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
```
Many Streaming Multiprocessors
Limited flexibility
in general GPUs have their own memory
High latency, high throughput
architecture keeps changing (e.g. direct NVLink, etc.)
Different strategies and language extensions to program GPUs

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

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