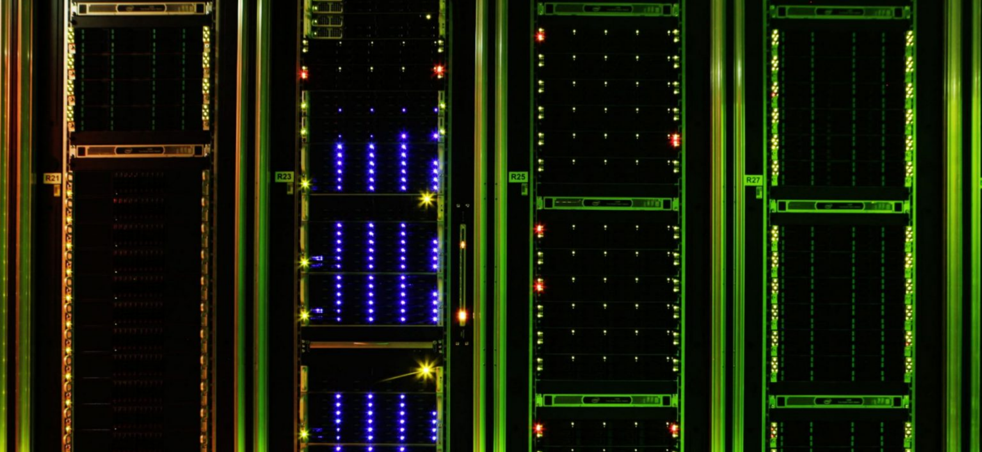
EPFL



Improving ORB5 Fourier field-solver parallel-scalability using partial DFTs and MPI derived types

Emmanuel Lanti, Thomas Hayward-Schneider

HPC-ACH annual meeting

15th of November 2023

SCITAS

Table of Contents

- Introduction to the ORB5 code
- Problem with the current Fourier solver
- Local DFTs and Fourier filter
- Zero copy DFTs
- Conclusions and possible future works

Introduction to the ORB5 code

ORB5 is:

- a global, gyrokinetic, electromagnetic, multi-species code use to study turbulent transport in the core
- based on a Lagrangian variational description, using a particle-in-cell (PIC) algorithm and a finite element representation of the fields

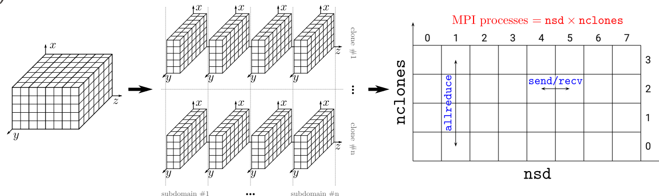
Introduction to the ORB5 code

ORB5 is:

- a global, gyrokinetic, electromagnetic, multi-species code use to study turbulent transport in the core
- based on a Lagrangian variational description, using a particle-in-cell (PIC) algorithm and a finite element representation of the fields

It is written in Fortran and parallelized using:

- 2D MPI decomposition:
  - Domain decomposition
  - Domain cloning
- OpenMP for multithreading support
- OpenACC for GPU support
- (OpenMP offload for GPU support)



ORB5 uses a 2D Fourier representation of the Poisson and Ampère equations:

$$A\phi = b \quad \Longrightarrow \quad \mathcal{F}A\mathcal{F}^{-1}\mathcal{F}\phi = \mathcal{F}b \tag{1}$$

- Because of the toroidal axisymmetry the equivalent system is composed of independent equations for the toroidal n modes
- Because the modes of interest are mainly aligned with the magnetic field only a small subset of poloidal m modes are required per n modes:

$$m \in \{-nq(s) - \Delta m, -nq(s) + \Delta m\} \tag{2}$$

- We typically use $\Delta m = 5$ resulting in keeping 11 m modes for each n mode

Problem with the current Fourier solver

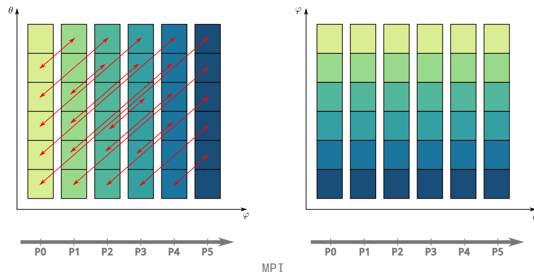
DFTs on distributed data

- To perform the DFTs, the transformed dimension needs to be contiguous
- Because of the domain decomposition, we need to parallel-transpose the data:
  - $(\theta, s, \varphi^p) \to \text{DFT}(\theta) \to (m, s, \varphi^p)$
  - $(m, s, \varphi^p) \to // \text{ transpose} \to (\varphi, s, m^p)$
  - $(\varphi, s, m^p) \to \text{DFT}(\varphi) \to (n, s, m^p)$
  - $(n, s, m^p) \to // \text{ transpose}(\theta) \to (m, s, n^p)$
- Same applies for the inverse DFT

Problem with the current Fourier solver

The pptransp algorithm

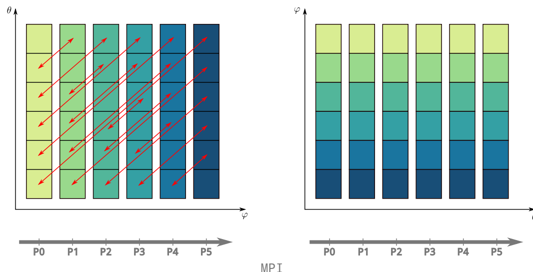
- The current parallel algorithm is called `pptransp`
- It uses `mpi_sendrecv` to exchange the distributed blocks
- Then, performs the transposes locally



Problem with the current Fourier solver

The pptransp algorithm

- The current parallel algorithm is called `pptransp`
- It uses `mpi_sendrecv` to exchange the distributed blocks
- Then, performs the transposes locally



- Since the GPU porting of the particle part, the parallel transpose is the main bottleneck
- Previous work from T. Ribeiro: *NEMOFFT project: Improved Fourier algorithms for global electromagnetic gyrokinetic simulations*, 2013

EPFL

Local DFTs and Fourier filter

Exploiting the filter through local DFTs

Building the RHS of Poisson and Ampère equations

The current implementation works as follows:

- Compute the poloidal DFT (constant factor left out for the sake of simplicity):

$$\hat{\rho}_{jl}^{m} = \sum_{k=0}^{N_\theta - 1} \rho_{jkl} \exp\left(\frac{-2\pi\imath km}{N_\theta}\right), \forall m \tag{3}$$

- Parallel transpose
- Compute the toroidal DFT :

$$\hat{\hat{\rho}}_{j}^{mn} = \sum_{l=0}^{N_\varphi - 1} \hat{\rho}_{jl}^{m} \exp\left(\frac{-2\pi\imath ln}{N_\varphi}\right), \forall n \tag{4}$$

- Apply the mode filter to keep only:
  - $n \in \{n_{\mathbf{min}}, n_{\mathbf{max}}\}$
  - $m \in \{-nq(s) - \Delta m, -nq(s) + \Delta m\} \cap \{m_{\mathbf{min}}, m_{\mathbf{max}}\}, \forall n$
- Both DFTs are computed using the FFTW library
- Parallel transpose again

New implementation: compute the toroidal DFTs locally and combine with the mode filter:

- Compute the poloidal DFT (same as original implementation):

$$\hat{\rho}_{jl}^{m} = \sum_{k=0}^{N_\theta - 1} \rho_{jkl} \exp\left(\frac{-2\pi\imath km}{N_\theta}\right), \forall m \tag{5}$$

New implementation: compute the toroidal DFTs locally and combine with the mode filter:

- Compute the poloidal DFT (same as original implementation):

$$\hat{\rho}_{jl}^{m} = \sum_{k=0}^{N_\theta - 1} \rho_{jkl} \exp\left(\frac{-2\pi\imath km}{N_\theta}\right), \forall m \qquad (5)$$

- Compute the toroidal DFT by splitting it "locally" to each subdomain and using the filter:

$$\hat{\hat{\rho}}_{j}^{mn} = \sum_{l=0}^{N_\varphi - 1} \hat{\rho}_{jl}^{m} \exp\left(\frac{-2\pi\imath ln}{N_\varphi}\right) = \sum_{\mathcal{S}} \sum_{l=0}^{N_\varphi^p - 1} \hat{\rho}_{jl_{\mathbf{glob}}}^{m} \exp\left(\frac{-2\pi\imath l_{\mathbf{glob}} n}{N_\varphi}\right) \qquad (6)$$

  Here only the modes $n \in \{n_{\min}, n_{\max}\}$ are computed and with $m \in \{-nq(s) - \Delta m, -nq(s) + \Delta m\} \cap \{m_{\min}, m_{\max}\}$

- The sum over the subdomains is done using `mpi_reduce_scatter`

Exploiting the filter through local DFTs

Main differences between the algorithms

The original `pptransp` algorithm:

- Two parallel transposes
- Arrays of size $\left(N_\theta, N_s, N_\varphi^p\right)$
- Allows intuitive implementation of RHS building

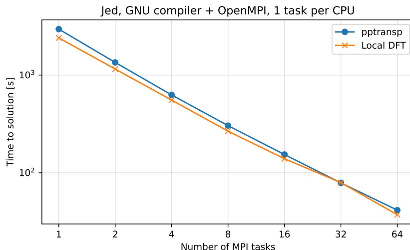
The "local DFT" algorithm:

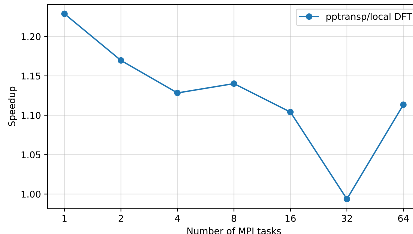
- One reduce scatter
- Arrays of size $(2\Delta m + 1, N_s, n_{\mathrm{max}} - n_{\mathrm{min}} + 1)$
- Much more complex implementation compared to `pptransp`
- May not work for certain signal/noise diagnostics

Exploiting the filter through local DFTs

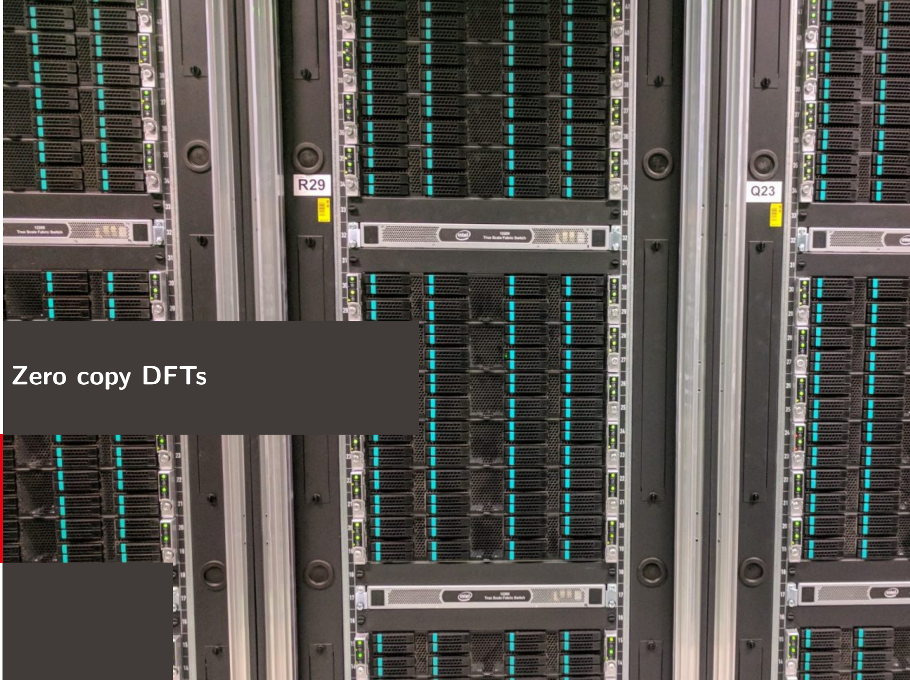
Initial scaling

- First scaling on the Jed cluster (EPFL)
  - 2 Intel Platinum (36 cores @ 2.4GHz), 2x25GB/s Ethernet network
  - with GNU 11.3 + OpenMPI 4.1.3
- Grid $(N_\theta, N_s, N_\varphi) = (512, 1024, 256)$, $n_{\min} = 0$, $n_{\min} = 64$, $\Delta m = 5$ $((11, 1024, 65))$
- One task per CPU to mimic the GPU setup
- Up to 23% faster (32 tasks case may be due to the machine)





EPFL



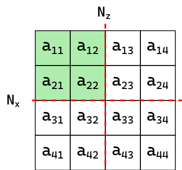
Zero copy DFTs

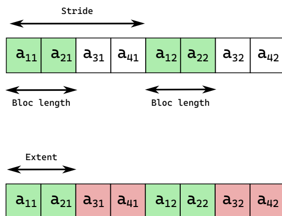
Using MPI derived types to avoid buffer copies

A 2D example

- One can use MPI derived types to perform a parallel transpose using a single call to `mpi_alltoall`
- Depending on the implementation, this could avoid 4 pack/unpack operations
- Use resized `mpi_type_vector`

Send type

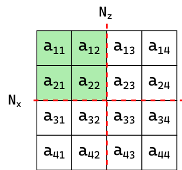


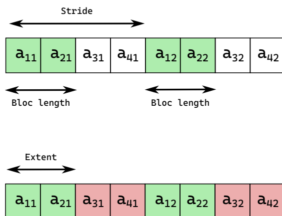


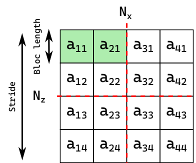
Using MPI derived types to avoid buffer copies

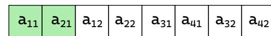
A 2D example

- One can use MPI derived types to perform a parallel transpose using a single call to `mpi_alltoall`
- Depending on the implementation, this could avoid 4 pack/unpack operations
- Use resized `mpi_type_vector`









Using MPI derived types to avoid buffer copies

A 2D example

In practice, we get the following code:

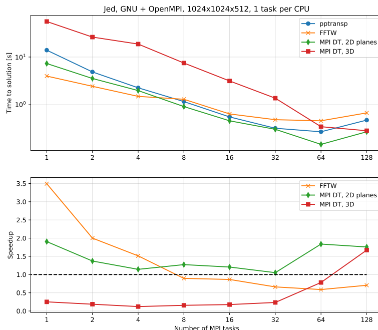
```
subroutine init_2d()
  lower_bound = 0
  call mpi_type_vector(Nzloc, Nxloc, Nx, MPI_DOUBLE_COMPLEX, mpi_vec_tmp, ierr)
  extent = Nxloc*sizeof(dble_cmplx)
  call mpi_type_create_resized(mpi_vec_tmp, lower_bound, extent, mpi_send_2d, ierr)
  call mpi_type_commit(mpi_send_2d, ierr)

  call mpi_type_vector(Nxloc, 1, Nz, MPI_DOUBLE_COMPLEX, mpi_vec_tmp, ierr)
  extent = sizeof(dble_cmplx)
  call mpi_type_create_resized(mpi_vec_tmp, lower_bound, extent, mpi_vec_tmp_resized, ierr)
  call mpi_type_contiguous(Nzloc, mpi_vec_tmp_resized, mpi_recv_2d, ierr)
  call mpi_type_commit(mpi_recv_2d, ierr)
end subroutine init_2d
```

```
subroutine transpose_2d(comm, array, array_t)
  type(TParallel), intent(in) :: comm
  complex(dp), dimension(:, :), intent(inout) :: array, array_t

  integer :: ierr

  call mpi_alltoall(array, 1, mpi_send_2d, array_t, 1, mpi_recv_2d, comm%id, ierr)
end subroutine transpose_2d
```

Using MPI derived types to avoid buffer copies

First scalings

- Scalings made on the EPFL Jed cluster with 1 MPI task per CPU, on a 1024x1024x512 grid
- Both GNU + OpenMPI (and Intel + Intel MPI have been tested)
- Could not reach the 512 task limit... WIP



Conclusion and possible future works

- Local DFT algorithm implemented and tested in ORB5
- First scaling shows that its performance can outperform `pptransp` (up to 20% in this case)
- Improve parallel transpose using MPI derived types
- MPI implementation seems to matter, more timings needed!

- Properly profile the code and further optimize
- Make scalings on as many machines and with as many software stacks
  - Implement some kind of scheduler to chose the most optimal solution
- Other optimizations can be added (see T. Ribeiro, 2013)