Parallelism and computational performance in GPAW: Recent features

Ask Hjorth Larsen

CAMD, Technical University of Denmark

June 1, 2021

- Many changes since 2016 Jyväskylä GPAW meeting
- In this talk: Brief description of overall parallelization and redistributions
- Then overview of many smaller (but significant) performance features

Ultrabrief overview of parallel distribution

```
GPAW(kpts=..., nbands=...,
     parallel={'kpt': K, 'domain': D, 'band': B})
```

- FD mode: $\tilde{\psi}_n^k(\mathbf{r})$: Distributed over k-points/spins, domains, bands
- PW mode: Like FD, but "domains" means a distribution over planewaves
- LCAO mode: Like FD, but "bands" often means "orbitals"
- Wavefunctions are the biggest and most expensive, and are generally shared among all processes in some way.
- Other quantities are sometimes stored redundantly for convenient access together with wavefunctions.
- Computations with redundantly stored data is often optimized using a "distribute—work—redistribute" pattern.

Ultrabrief overview of parallel distribution

3D main CPU mesh

- k-points/spins $\psi_{\underline{\underline{k}}n}(\mathbf{r})$
- bands/orbitals $\psi_{k\underline{\underline{n}}}(\mathbf{r})$, $H_{\underline{\underline{\mu}}\nu}$, $c_{\mu\underline{\underline{n}}}$
- Domains $\psi_{kn}(\underline{\underline{\mathbf{r}}})$
- Actual news (since 2016): Jens Jørgen Mortensen added "domains" (distribution over planewaves) to PW mode!

Temporary redistributions to "world"

- ScaLAPACK $H_{\underline{\underline{\mu\nu}}}$, $c_{\underline{\underline{\mu n}}}$
- Atomic quantities ΔH_{asp}
- Fine-grid (Poisson, XC)
- LCAO atomic corrections/projections: Now with sparse Scipy matrices!

3D grid redistribution

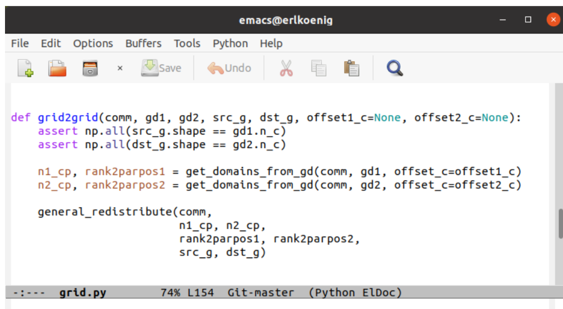
emacs@erlkoenig

File Edit Options Buffers Tools Python Help

Save Undo

```
def redistribute(gd, gd2, src, distribute_dir, reduce_dir, operation='forth'):
    """Perform certain simple redistributions among two grid descriptors.

    Redistribute src from gd with decomposition X x Y x Z to gd2 with
    decomposition X x YZ x 1, or some variation of this.  We say that
    we "reduce" along Z while we "distribute" along Y.  The
    redistribution is one-to-one.

             ____________                          ____________
    i       /     /     /|          r             /  /  /  /  /|
    n      /_____/_____/ |         i             /  /  /  /  / |
    d     /     /     /| |        d             /  /  /  /  /  |
    e    /_____/_____/ | j          forth      /__/__/__/__/   j
    p    |     |     | |/|      e   ------->   |  |  |  |  |  /|
    e    |     |     | ł |     c   <-------    |  |  |  |  | / |
    n    |_____|_____|/| j    u      back      |__|__|__|__|/  j
    d    |     |     | |/    d                 |  |  |  |  |  /
    e    |     |     | /    e                  |  |  |  |  | /
    n    |_____|_____|/    r                   |__|__|__|__|/
    t

         d i s t r i b u t e   d i r

    Directions are specified as 0, 1, or 2.  gd2 must be serial along
    the axis of reduction and must parallelize enough over the
    distribution axis to match the size of gd.comm.

    Returns the redistributed array which is compatible with gd2.

    Note: The communicator of gd2 must in general be a special
    permutation of that of gd in order for the redistribution axes to
    align with domain rank assignment.  Use the helper function
    get_compatible_grid_descriptor to obtain a grid descriptor which
    uses a compatible communicator."""
U:---  grid_redistribute.py    3% L49    Git-master  (Python ElDoc)
```

(Old version; work presented in 2016 GPAW meeting)



- More general version for any grids and any communicators
- Simple arguments: Function is very easy to call
- Many parts of the code do not yet make good use of grid redistribution
- Used for: FastPoissonSolver (Mikael Kuisma), extra vacuum Poisson solver (Tuomas Rossi), augment_grids/libvdwxc

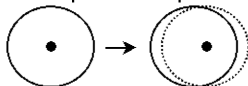
Use "augment grids" in non-small systems!

Use `GPAW(parallel=dict(augment_grids=True), ...)`

```
Total number of cores used: 6
Parallelization over k-points: 6
Domain decomposition: 1 x 1 x 1
                      3 x 2 x 1 (xc only)

Number of atoms: 16
Number of atomic orbitals: 144
Number of bands in calculation: 110
Number of valence electrons: 176
Bands to converge: occupied
```

Reusing wavefunctions when positions change

If atoms move a little bit, it's a good idea to reuse the wavefunctions from the previous positions.



Methods for reusing wavefunctions

- Do nothing (keep wavefunctions unchanged). Fine for LCAO where basis functions automatically "follow" atoms.
- Some DFT codes use extrapolation from previous positions (e.g. for MD)
- GPAW FD/PW: Un-project and re-project wavefunctions to new positions

PAW projector/partial wave dual basis

Consider the good old PAW transformation:

$$|\tilde{\psi}_n\rangle = \hat{\mathcal{T}} |\psi_n\rangle , \tag{1}$$

which is defined from partial waves and projectors:

$$\hat{\mathcal{T}} = 1 + \sum_{ai} (|\tilde{\phi}_i^a\rangle - |\phi_i^a\rangle) \langle \tilde{p}_i^a| . \tag{2}$$

Partial waves and projectors form a dual basis which is approximately complete close to their atom:

$$\sum_i |\tilde{\phi}_i^a\rangle \langle \tilde{p}_i^a| \approx \text{identity (close to atom } a) \tag{3}$$

Wavefunction reuse

- Subtract projected partial waves from wavefunctions:

$$|\tilde{\psi}_n\rangle \leftarrow |\tilde{\psi}_n\rangle - \sum_{ai} |\tilde{\phi}_i^a\rangle_{\mathbf{R}^a_{\mathrm{old}}} \langle \tilde{p}_i^a|\tilde{\psi}_n\rangle \quad (4)$$

- Move each atom from $\mathbf{R}^a_{\mathrm{old}}$ to $\mathbf{R}^a_{\mathrm{new}}$, assuming projections $P^a_{ni} = \langle \tilde{p}_i^a|\tilde{\psi}_n\rangle$ remain the same
- Re-add projected partial waves to wavefunctions around new atomic centers:

$$|\tilde{\psi}_n\rangle \leftarrow |\tilde{\psi}_n\rangle + \sum_{ai} |\tilde{\phi}_i^a\rangle_{\mathbf{R}^a_{\mathrm{new}}} \langle \tilde{p}_i^a|\tilde{\psi}_n\rangle \quad (5)$$

- (Requires application of $\mathbf{k}$-point phase $\exp(i\mathbf{k}\cdot\mathbf{\Delta R})$ to coefficients P^a_{ni} for any atom which moves across cell boundary)

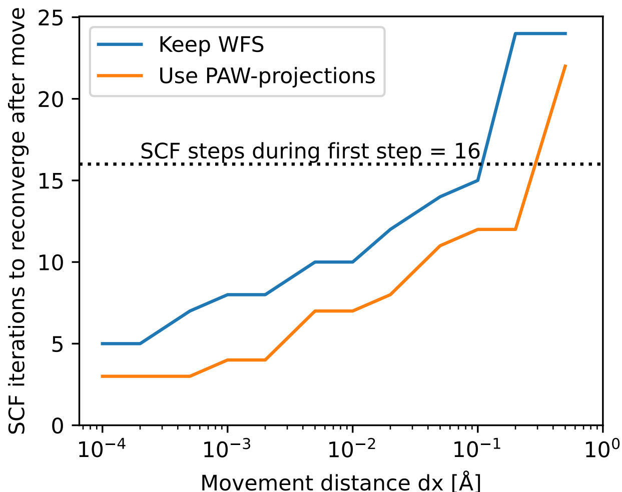
```
GPAW(experimental={'reuse_wfs_method': 'paw'}, ...)
```

With method 'keep':

```
                        log10-error:     total
           time         wfs      density  energy
iter:   1  21:49:44     +0.45             -91.127478
iter:   2  21:49:47     -0.77    -1.09    -77.999287
iter:   3  21:49:50     -0.92    -1.23    -71.599034
iter:   4  21:49:54     -0.87    -1.41    -70.405095
```

With method 'paw':

```
                        log10-error:     total
           time         wfs      density  energy
iter:   1  21:18:58     -1.09             -71.425455
iter:   2  21:19:02     -2.55    -1.69    -69.986275
iter:   3  21:19:05     -1.66    -1.85    -68.960482
iter:   4  21:19:09     -2.97    -2.40    -68.943224
```



(Note: larger `dx` eventually increase energy which makes system harder to converge)

LCAO-based WFS reuse method

$$|\tilde{\psi}_n\rangle \leftarrow |\tilde{\psi}_n\rangle \pm \sum_{\mu\nu} |\Phi_\mu\rangle \, S^{-1}_{\mu\nu} \, \langle \Phi_\nu | \psi_n \rangle \tag{6}$$

Pros:

- Handles orthogonality correctly
- Supports more complete basis sets

Cons:

- Method is likely too slow to be useful
- Somewhat more complex due to inversion/linsolve
- Not implemented/working in all cases ($\mathbf{k}$-points)

But more important: Someone should implement a way to reuse wavefunctions when the cell changes!

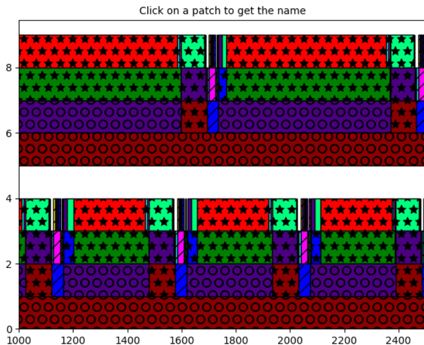
ELPA

- ELPA is an efficient parallel eigensolver library
- `https://elpa.mpcdf.mpg.de/`
- A. Marek *et al* 2014 J.Phys.: Condens. Matter **26** 213201
- In GPAW, Elpa can now be used together with ScaLAPACK and uses the same parallel data distribution
- Elpa solves generalized eigenvalue problem about twice as fast as ScaLAPACK/DC (in our benchmark)
- Elpa can also do other operations; we can probably benefit from exploring those!
- `GPAW(parallel=dict(sl_auto=True, use_elpa=True), ...)`

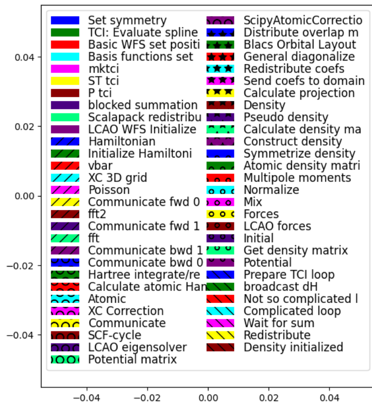
```
from gpaw.utilities.timing import ParallelTimer
calc = GPAW(timer=ParallelTimer(), ...)
...
```

```
$ gpaw-plot-parallel-timings timings.*.txt
```

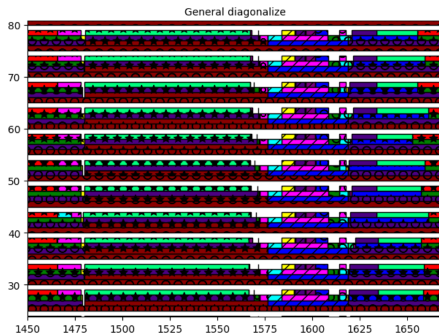
Parallel timings: Elpa and ScaLAPACK



Call stack as a function of time (seconds) for Elpa (bottom) and ScaLAPACK/DC (top)



Comparing multiple ranks



- Timings plotted for multiple ranks in same computation
- Only minor discrepancies (teal = scalapack redistribution)
- Further optimization and load balancing can be quite important for systems with fewer atoms, e.g. MD simulations

Broadcast imports

Module initialization in Python

- Locate file on disk, often searching multiple import paths
- Read file (bytecode)
- Register module in `sys.modules` and execute module code

Parallel bottlenecks with HPC

- Many cores need to read each file at the same time
- Python/GPAW/ASE/numpy/scipy contain hundreds of modules
- BlueGene/P: Tens of thousands of cores, import overhead can be more than an hour
- Inefficient network filesystems: I have seen 1 minute import overhead on just one 12-core node

Broadcast imports

```
from gpaw.broadcast_imports import broadcast_imports

with broadcast_imports:
    import numpy as np
    import scipy
    ...
```

- GPAW uses broadcast imports to import (parts of) itself and other libraries
- Users can also use broadcast imports if/whenever they want
- Essential for massively parallel computations and exascale

Broadcast imports

What happens when we enter the context (“`with:`” block)?

On `rank == 0`	On `rank != 0`
Set custom importloader	Set custom importloader
Create empty “module cache”	Wait for data ...
On import, store module bytecode	
Execute code in `with:` block	
Broadcast bytecode	Receive bytecode
Restore default importloader	Execute code in `with:` block
	On import, store module in `sys.modules` and execute bytecode
	Restore default importloader

Important to restore default loader: Else the code may import a module only on a subset of ranks, causing deadlock

Conclusions

Novel, interesting, and/or underused features

- Wavefunction reuse using PAW projectors
- Augment grids: Use all cores for XC
- grid2grid: Distribute directly from 3D grid into other 3D grid
- FastPoissonSolver: We no longer need to worry about Poisson solvers!
- Elpa/ScaLAPACK (mostly for LCAO)
- Dev: Benchmark your features with ParallelTimer
- Broadcast imports for big computations

How can we best leverage these features, and document or automate parameter choices?