BERTHA and PyBERTHA: state of the art for full four-component Dirac-Kohn-Sham calculations

Loriano Storchi , Matteo De Santis, Leonardo Belpassi

University of Chieti-Pescara, University of Perugia, ISTM-CNR

ParCo2019

Table of contents

t's try it anyway...

$$(i\gamma^\mu \partial_\mu - m)\psi = 0$$

Introduction

mehow, want this, but $i\gamma^\mu$ not four

Introduction

It is universally recognized that relativistic effects play a crucial role in chemistry, especially for heavy elements

- The challenge clearly arises from the fact that heavy elements have a very large number of electrons, and both relativistic effects and electron correlation play a crucial role
- A particularly suitable and promising theoretical framework to appropriately treat these systems is the Dirac-Kohn-Sham model (DKS)

Introduction

The DKS equation reads:

$$\left[c\begin{pmatrix} 0 & \boldsymbol{\sigma} \\ \boldsymbol{\sigma} & 0 \end{pmatrix}\boldsymbol{p} + \begin{pmatrix} \boldsymbol{I} & 0 \\ 0 & -\boldsymbol{I} \end{pmatrix}c^2 + v^{(1)}(\boldsymbol{r})\right]\psi_i(\boldsymbol{r}) = \varepsilon_i\psi_i(\mathbf{r})$$

Where there is the speed of light in vacuum, the electron four-momentum and the Pauli 2x2 spin matrices

Introduction

As for the nonrelativistic case the potential term is made of: nuclear potential, Coulomb interaction and the exchange-correlation term:

$$\left(i\gamma^{\mu}\partial_{\mu} - m\right)\psi = 0$$

Where clearly in the last two terms the relativistic electronic density appear

A proper and genuine expression for the exchange-correlation functional is still missed

Introduction

The four-spinor solution of the DKS equation is expressed in BERTHA as a linear combination of 2N G-spinor basis functions:

$$\psi_i(\boldsymbol{r}) = \begin{bmatrix} \sum_{\mu=1}^{N} c_{\mu i}^{(\mathrm{L})} M_{\mu}^{(\mathrm{L})}(\boldsymbol{r}) \\ i \sum_{\mu=1}^{N} c_{\mu i}^{(\mathrm{S})} M_{\mu}^{(\mathrm{S})}(\boldsymbol{r}) \end{bmatrix}$$

Due to presence of the Large and Small components the basis set is doubled with respect to a non-relativistic or a two-component calculation.

Where L identify the large component and S the small one

Introduction

Finally the matrix representation of the DKS operator in the G-spinor basis is (**Complex arithmetic**):

$$\begin{bmatrix} \boldsymbol{v}^{(\mathrm{LL})} + \boldsymbol{J}^{(\mathrm{LL})} + \boldsymbol{K}^{(\mathrm{LL})} + mc^2\boldsymbol{S}^{(\mathrm{LL})} & c\boldsymbol{\Pi}^{(\mathrm{LS})} \\ c\boldsymbol{\Pi}^{(\mathrm{SL})} & \boldsymbol{v}^{(\mathrm{SS})} + \boldsymbol{J}^{(\mathrm{SS})} + \boldsymbol{K}^{(\mathrm{SS})} - mc^2\boldsymbol{S}^{(\mathrm{SS})} \end{bmatrix}$$

Coulomb potential

Exchange-correlation potential

Nuclear potential

Kinetic energy operator matrix

Overlap matrix

Introduction

The associated eigenvalue equation reads:

$$\left(i\gamma^{\mu}\partial_{\mu} - m\right)\psi = 0$$

Spinors expansion vectors

The matrix $\mathbf{H}_{DKS}$ depends, because of **J** and **K**, on the canonical spinor-orbitals produced by its diagonalization, so that the solution **c** must be obtained recursively to self-consistence.

Introduction

Which matrix depends on density which is not

$$v_{\mu\nu}^{(\mathrm{TT})} = \int v_{\mathrm{N}}(\boldsymbol{r})\rho_{\mu\nu}^{\mathrm{TT}}(\boldsymbol{r})\mathrm{d}\boldsymbol{r}$$

$$J_{\mu\nu}^{(\mathrm{TT})} = \int v_{\mathrm{H}}^{(1)}[\rho(\boldsymbol{r})]\rho_{\mu\nu}^{\mathrm{TT}}(\boldsymbol{r})\mathrm{d}\boldsymbol{r}$$

$$K_{\mu\nu}^{(\mathrm{TT})} = \int v_{\mathrm{XC}}^{(1)}[\rho(\boldsymbol{r})]\rho_{\mu\nu}^{\mathrm{TT}}(\boldsymbol{r})\mathrm{d}\boldsymbol{r}$$

$$S_{\mu\nu}^{(\mathrm{TT})} = \int \rho_{\mu\nu}^{\mathrm{TT}}(\boldsymbol{r})\mathrm{d}\boldsymbol{r}$$

$$\Pi_{\mu\nu}^{(\mathrm{TT}')} = \int M_{\mu}^{(\mathrm{T})\dagger}(\boldsymbol{r})(\boldsymbol{\sigma}\cdot\boldsymbol{p})M_{\nu}^{(\mathrm{T}')}(\boldsymbol{r})\mathrm{d}\boldsymbol{r}$$

where the sum runs over the occupied positive-energy states

G-spinor overlap densities

Introduction

- The matrix H_{DKS} depends, because of **J** and **K**, on the canonical spinor-orbitals produced by its diagonalization, so that the solution **c** must be obtained recursively to self-consistence.
- As in the nonrelativistic context, once a guess density has been provided (**usually cast as a superposition of atomic densities**),
- The problem formally reduces to the **evaluation (manly J and K) of the integrals in previous equations for the assembling of H_{DKS}** and the iterative solution of the **eigenvalue problem (diagonalization)** with up-to-date **J** and **K** integrals at each cycle.

Introduction

- The J and the K matrix construction scales respectively as $O(N^4)$ and $O(N^3)$ with respect the number of atoms.
- We take advantage of the of **density fitting** techniques for an efficient evaluation of the **Coulomb J and exchange-correlation K matrices**
- **The relativistic electronic density is expanded in a set of N_{aux} auxiliary atom-centered functions:**

$$\tilde{\rho} = \sum_{t=1}^{N_{aux}} d_t f_t(\boldsymbol{r})$$

Introduction

This allows for an efficient construction of the **J** and **K** matrices in a single step

$$\tilde{J}^{(\mathrm{TT})}_{\mu\nu} + \tilde{K}^{(\mathrm{TT})}_{\mu\nu} = \sum_{t=1}^{N_{\mathrm{aux}}} I^{(\mathrm{TT})}_{t,\mu\nu}(d_t + z_t)$$

Where the vectors **d** an **z** are the solution of two small and real $\mathbf{N_{AUX}}$ **X** $\mathbf{N_{AUX}}$ linear equation systems.

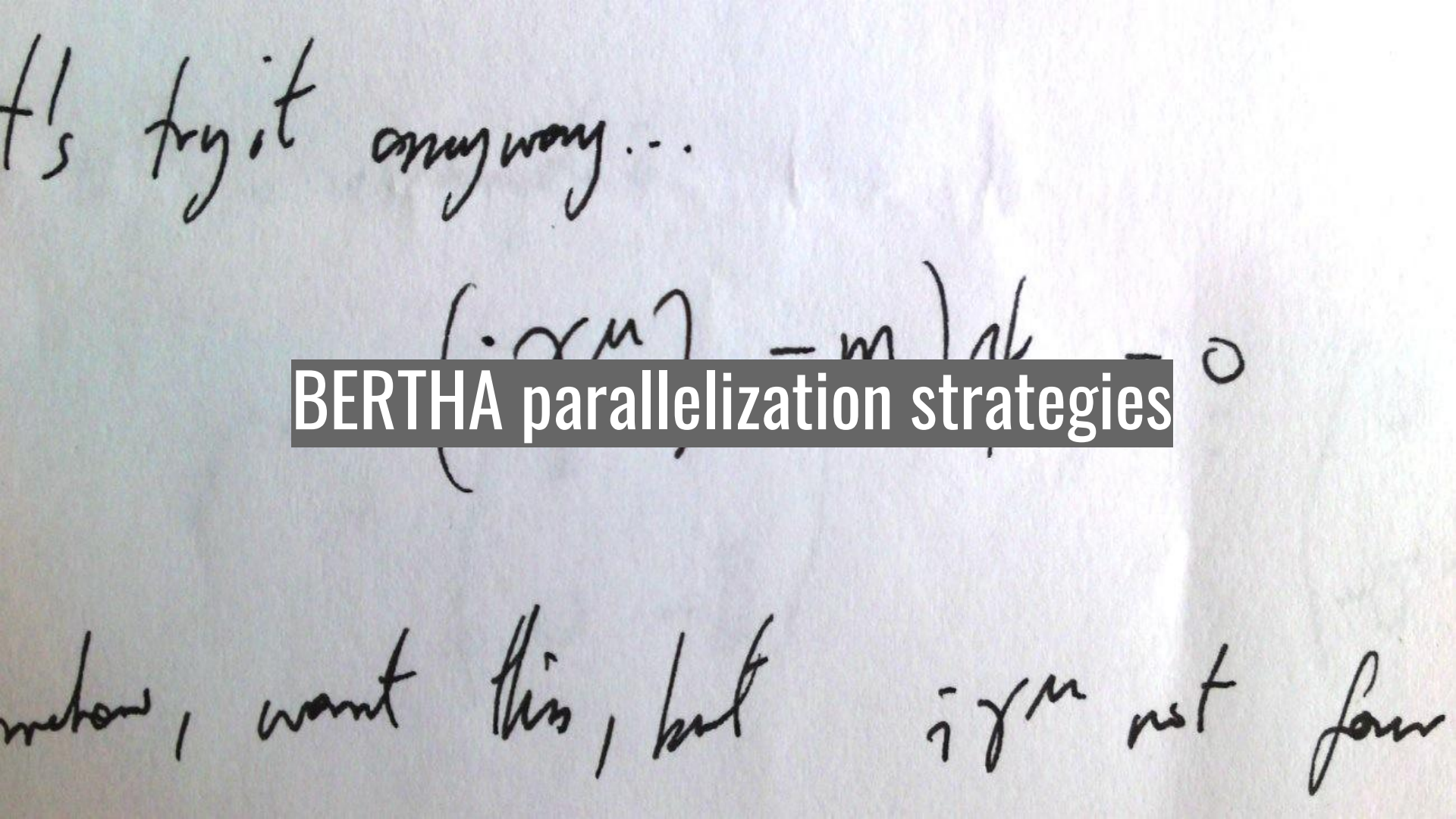
Introduction

How efficient is the density fitting approach ?

Cluster	DKS Size	$(J+K)_{conv}$	$(J+K)_{fit}$	Speed-up
Au_2	1560	$1.86 \cdot 10^3$	7.4	251
Au_4	3120	$1.71 \cdot 10^4$	44.1	388
Au_8	6240	$1.71 \cdot 10^5$	296	578
Au_{16}	12480	$1.91 \cdot 10^{6a}$	$2.16 \cdot 10^3$	884

[a] Extrapolated value.

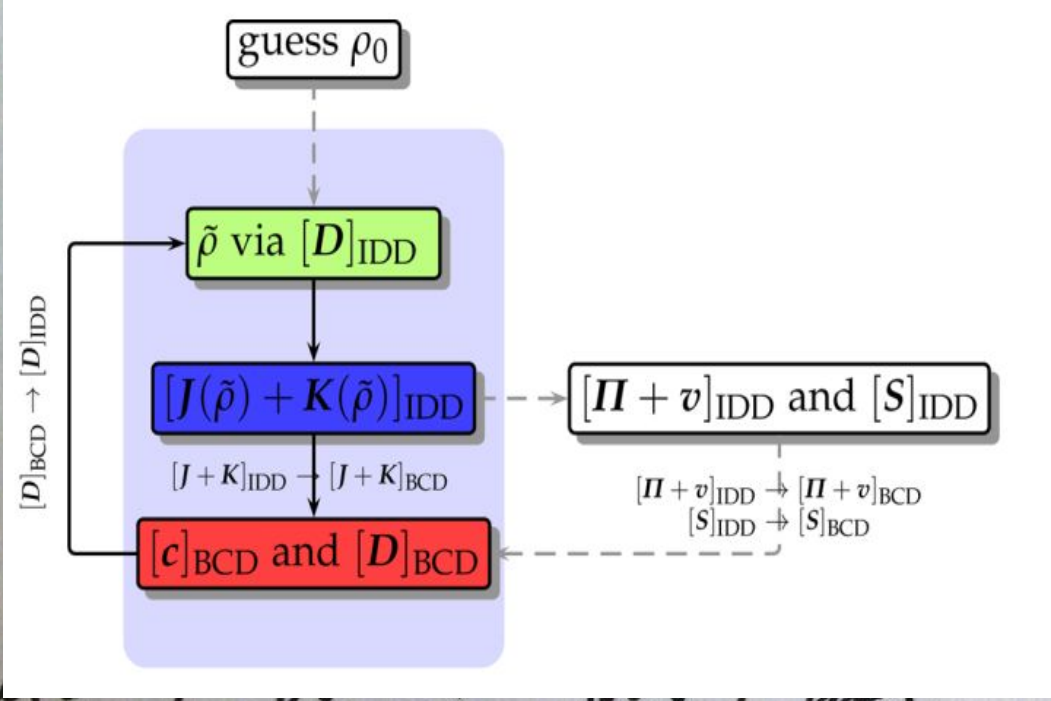
BERTHA parallelization strategies



Parallelization strategies

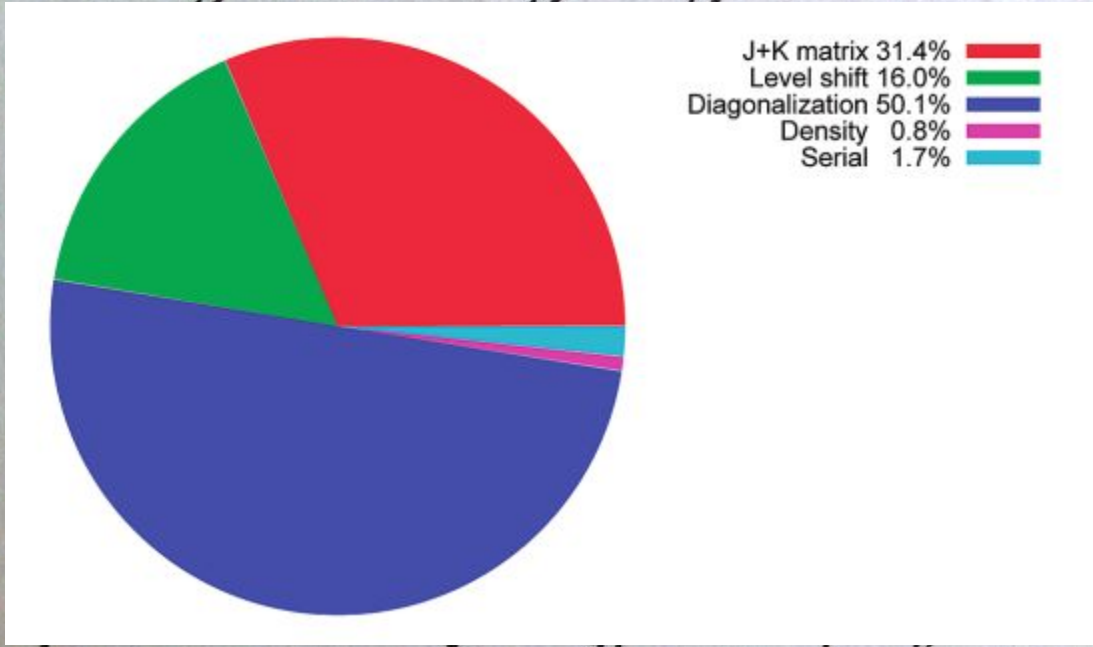
- According to Amdahl's law, serial portion of code limits the speedup, thus **we tried to remove any single potion os serial code**
- During the SCF procedure, in fact, **the "bulk" memory allocation is due to several 2N × 2N complex Hermitian matrices:**
  - the overlap matrix S
  - the one-electron matrix
  - the Coulomb plus exchange-correlation matrix J + K
  - the matrix of the eigenvectors
  - the density matrix

Parallelization strategies



A similar strategy has been adopted for the one electron and superposition matrices

Parallelization strategies



CPU time percentages for the various phases of a serial DKS calculation of the gold cluster Au_{16} . All linear algebra operations are performed with the Intel Math Kernel Library. We are here considering each SCF step

Parallelization strategies

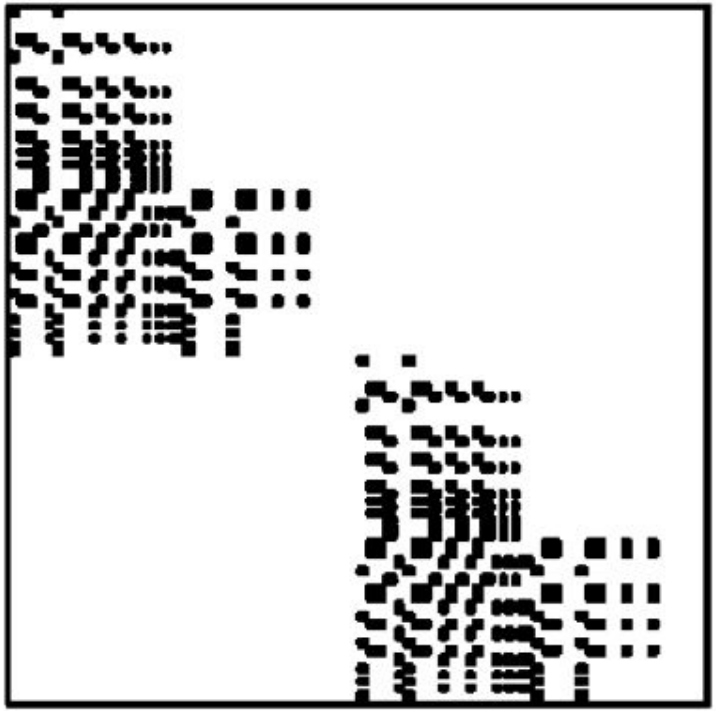
- ScaLAPACK has been used for all the linear algebra
  - P processes of a generic parallel execution are mapped onto a $P_r \times P_c$ two-dimensional "process grid"
  - Each dense matrix is then decomposed into blocks of suitable size according to a specific the so-called Block Cyclic Distribution (BCD)

- We clearly need some efficient way to distribute our matrices among the processes according to the BCD scheme, so that linear algebra operations can be carried out by ScaLAPACK routines in parallel

Parallelization strategies

- J + K matrix parallelization strategy (using MPI)
  - An efficient parallel construction of the matrix J + K has been achieved by cyclically assigning to each process the allocationand computation of blocks whose offsets and dimensions depend on the specific structure of the G-spinor matrices.
  - We will refer as to Integral Driven Distribution (IDD), is naturally dictated by the grouping of G-spinor basis functions in sets characterized by common origin and angular momentum

Parallelization strategies



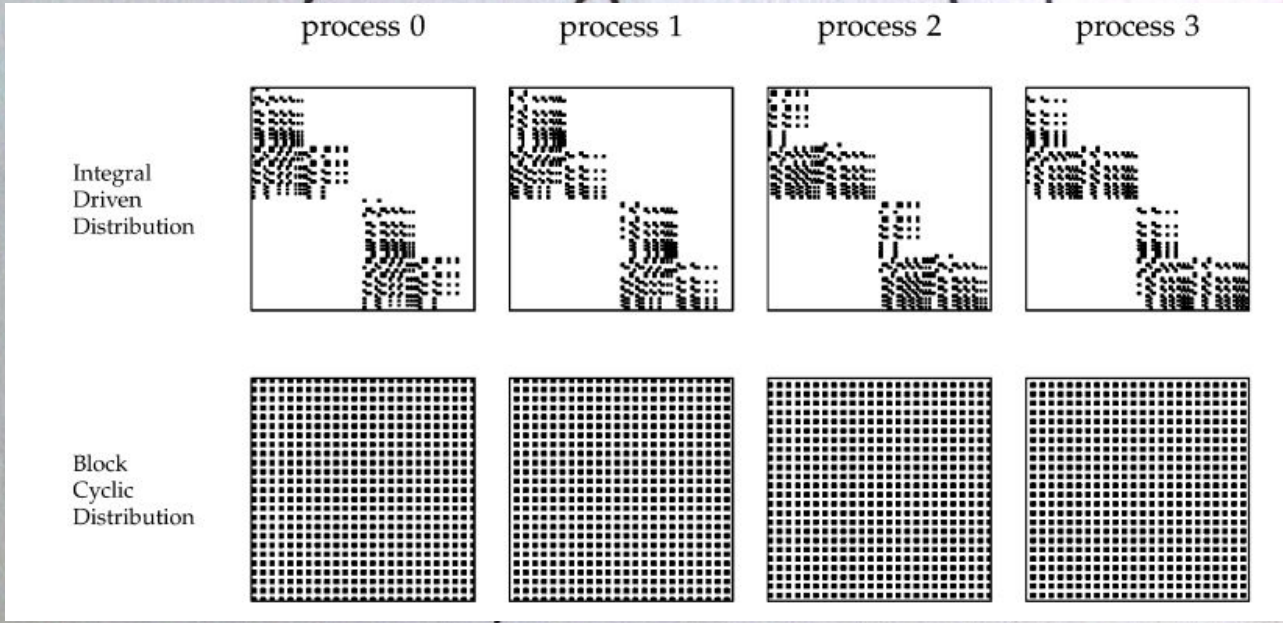
Integral Driven Distribution (IDD): Cyclically assigning to each process the allocation and computation of blocks whose offsets and dimensions depend on the specific structure of the G-spinor matrices.

```
DO block_num = 1, max_nom_of_block
  IF ((my_rank+((num_of_processors)*my_block_num)) == block_num)
    my_block_num = my_block_num + 1
    ALLOCATE block
    COMPUTE block
  END IF
ENDDO
```

Linear algebra using SCALAPACK

(Level Shift, Diagonalization, Density)

Parallelization strategies



Parallelization strategies

setup an “info” array

Allocates send buffer

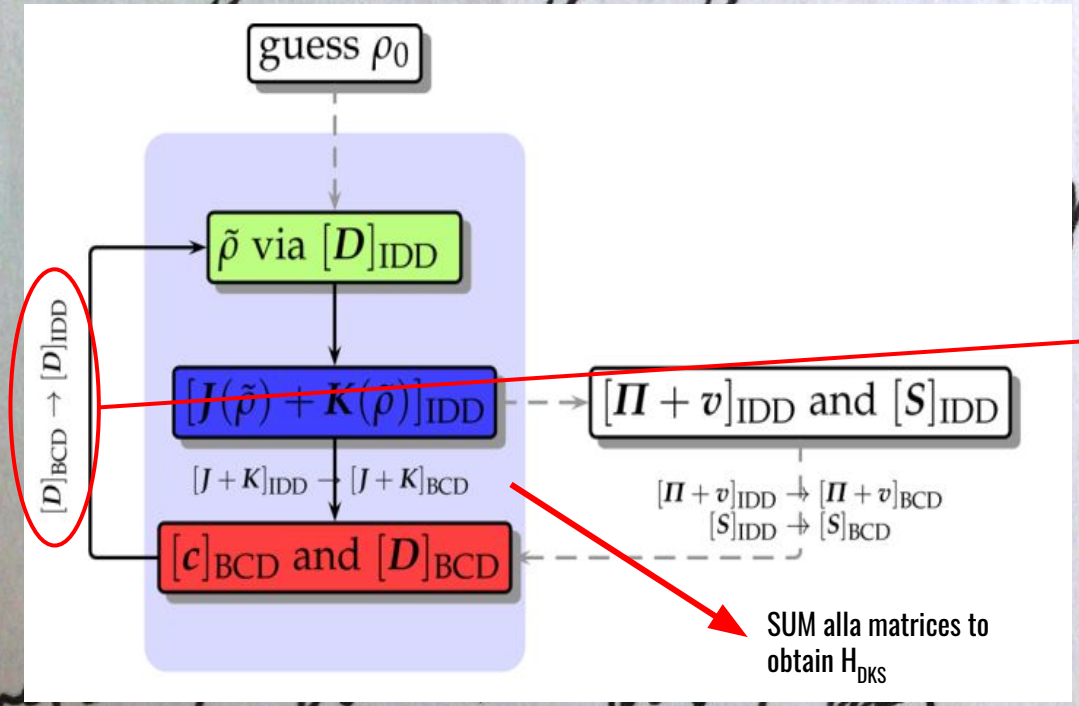
Pack the owned data (along with the related destination
Indices, so local indices) into the send-buffers

Communicate

Deallocate send buffer

```
loop on processes
    if my turn then
        make my 'info' array available to all and send packed data
    else
        allocate receive-buffers according to the 'info' array
        made available by the sender, receive and unpack data,
        deallocate receive-buffers
    end if
end loop
```

Parallelization strategies



SUM alla matrices to obtain H_{DKS}

A similar strategy has been adopted for the one electron and superposition matrices

In the IDD scheme, instead , the distribution is much less regular. A convenient and efficient representation is obtained using a derived data type, composed of a two-dimensional array and some metadata describing its size and placement in the global matrix. On each process, an array of such derived data types is then used to identify each local IDD block

Parallelization strategies

Wall-Clock Time in Seconds (Average over 4 SCF Cycles) Spent in the Distribution Mapping Routines during Calculation

	$[(Ph_3P)Au(C_2H_2)]^+$		Au_8		Au_{16}		Au_{32}	
P	$t_{BCD\to IDD}$	$t_{IDD\to BCD}$	$t_{BCD\to IDD}$	$t_{IDD\to BCD}$	$t_{BCD\to IDD}$	$t_{IDD\to BCD}$	$t_{BCD\to IDD}$	$t_{IDD\to BCD}$
4	0.70	0.57	0.74	0.60	3.52	2.46	20.61	9.68
8	0.39	0.36	0.41	0.38	1.85	1.46	9.09	6.22
16	0.21	0.25	0.22	0.24	0.96	0.98	5.96	3.72
32	0.20	0.24	0.21	0.25	0.73	0.87	2.95	3.32
64	0.35	0.38	0.36	0.40	0.82	1.00	2.55	3.27
128	(2.5%) 0.92	(2.3%) 0.86	(2.9%) 0.90	(2.8%) 0.88	1.50	1.64	3.23	4.02
256	(6.9%) 2.69	(6.6%) 2.56	(6.5%) 1.84	(7.4%) 2.11	(2.6%) 3.84	(2.4%) 3.64	6.10	6.69

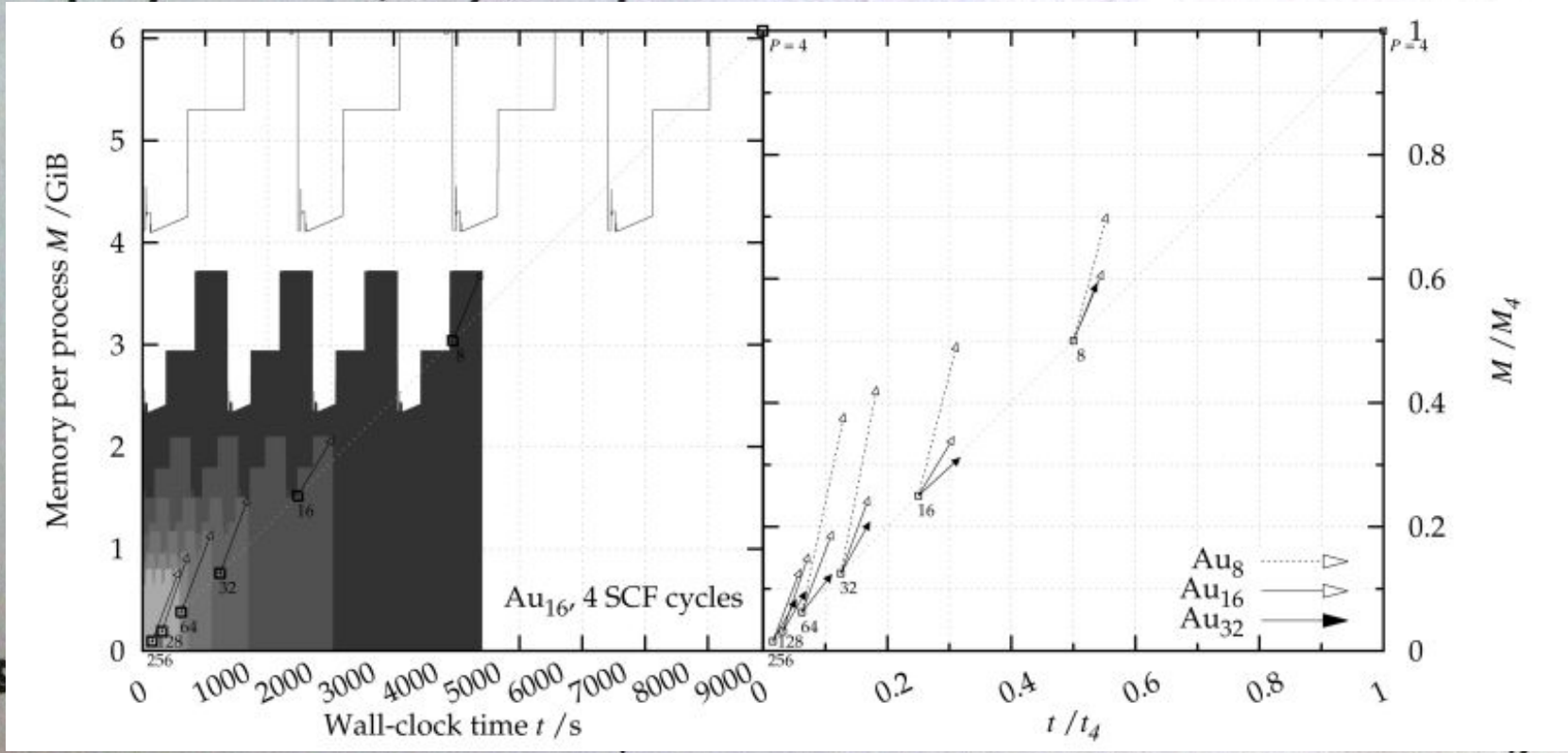
[a] Percentages of the SCF iteration times are smaller than 1% in any case except where otherwise noted in parentheses.

Parallelization strategies

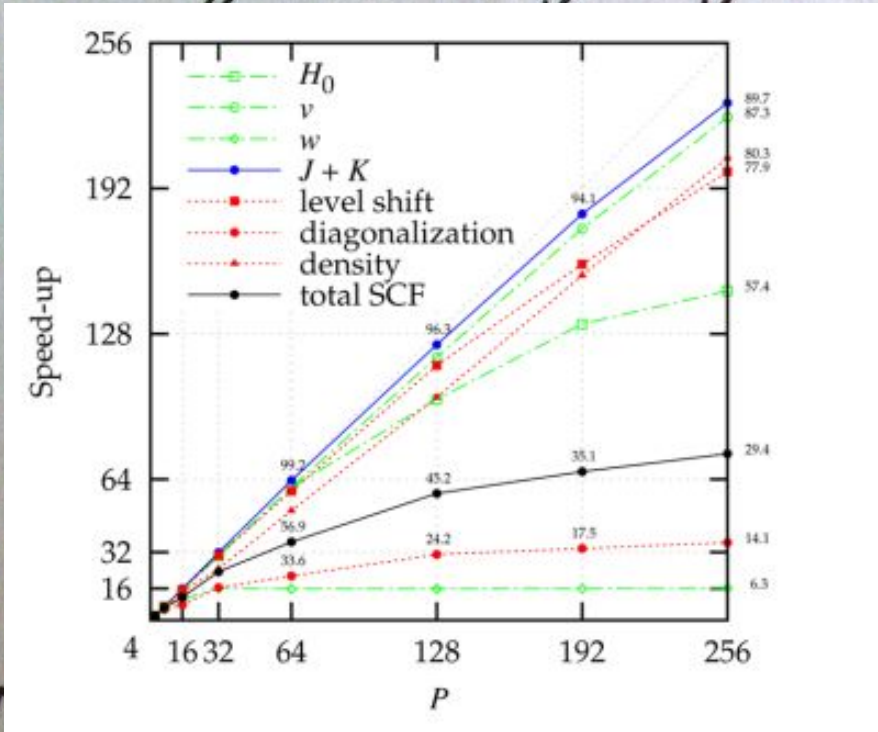
Memory Per Process Peak (Average Value M_{av}, Maximum Positive Δ_+ and Negative Δ_- Deviations) in MiB over P Processes

	$[(Ph_3P)Au(C_2H_2)]^+$			Au_8			Au_{16}			Au_{32}		
P	Δ_-	M_{av}	Δ_+	Δ_-	M_{av}	Δ_+	Δ_-	M_{av}	Δ_+	Δ_-	M_{av}	Δ_+
4	76	1842	37	34	1882	60	81	6214	46	126	23835	187
8	57	1253	59	64	1303	71	54	3770	90	88	14106	232
16	12	884	15	8	888	44	14	2124	23	8	7405	63
32	62	799	84	33	773	96	44	1499	85	53	4827	155
64	47	700	92	37	672	92	80	1167	80	64	2880	83
128	21	631	115	24	620	110	54	961	81	48	2220	101
256	10	599	134	15	586	119	52	866	83	80	1942	92

Parallelization strategies



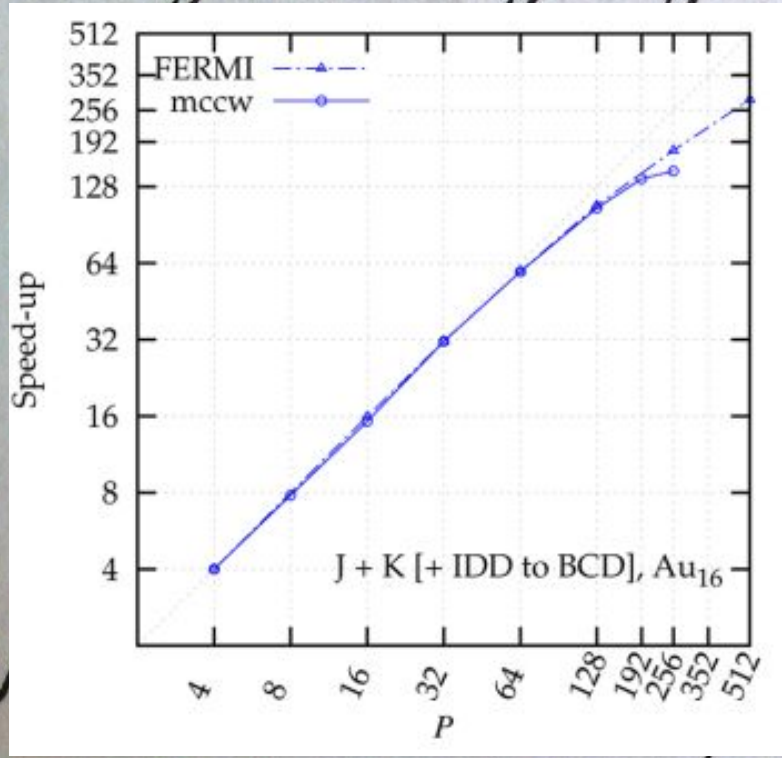
Parallelization strategies



Speedup for all of the computational kernels of the SCF procedure for Au_{32} as a function of the number of processors P

Mccw cluster equipped with Intel(R) Xeon(R) CPU E5-26700 2.60 GHz (24 nodes, 384 cores with 128 GiB/node, 8 GiB/core) and Infiniband network

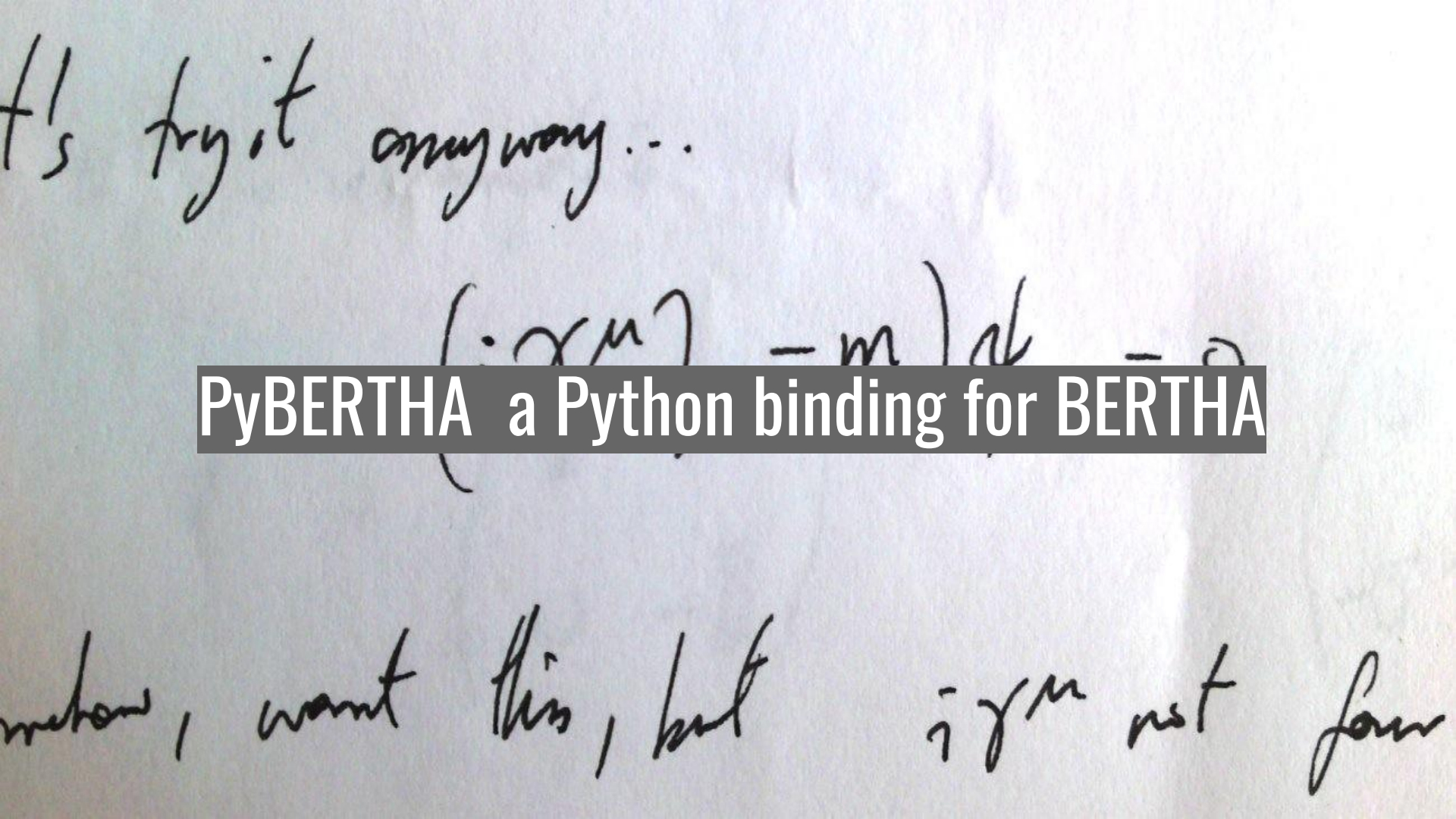
Parallelization strategies



Speedup for the J + K computation kernel for Au_{16} as a function of the number of processors P

FERMI located at CINECA, Italy and equipped with IBM PowerA2 1.6 GHz (10240 nodes, 163840 cores with 16 GiB/node, 1GiB/core) and a 11 links → 5D Torus network interface.

PyBERTHA a Python binding for BERTHA

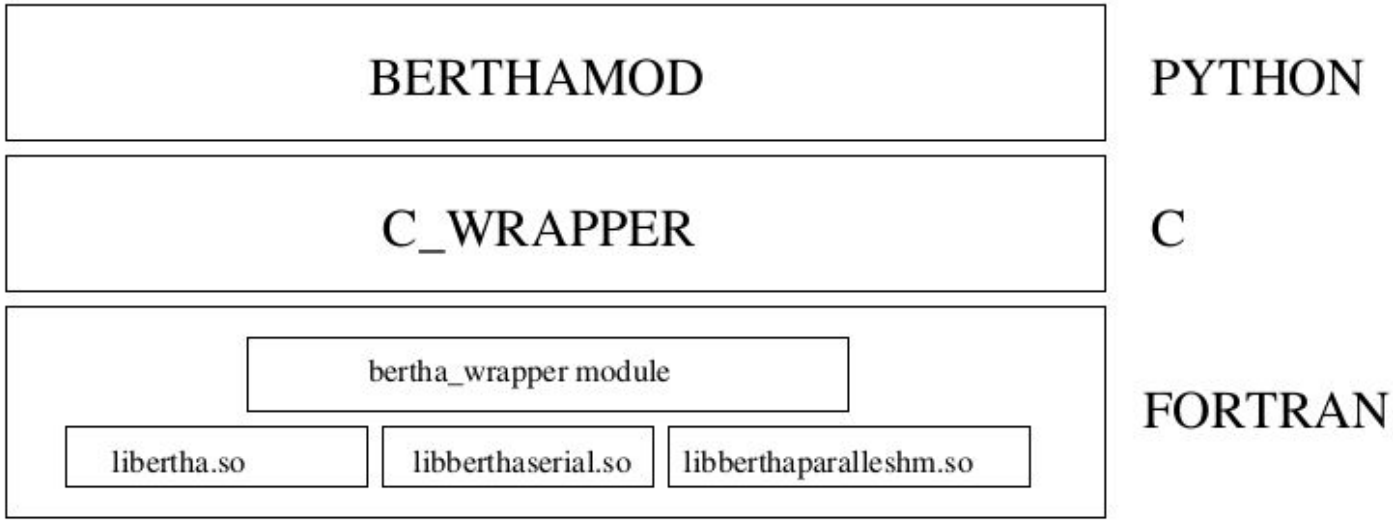


PyBERTHA a Python binding for BERTHA

- Undoubtedly the Python programming language is emerging as one of the most important and used HLL also in the field of scientific computing.
- Python HLL, besides providing an extensive range of modules to be used to solve comprehensive set of computational problems, enables for a quick prototyping
- So Python is clearly a natural choice for the BERTHA project.

PyBERTHA a Python binding for BERTHA

An overview of the software and HLL layers.



PyBERTHA a Python binding for BERTHA

```
module bertha_wrapper
use, intrinsic :: iso_c_binding
...
subroutine bertha_main(fittcoefffname, vctfilename, ovapfilename,
                       fittfname, eigen, ovap_ptr, eige_ptr,
                       fock_ptr)

implicit none

real (c_double) :: eigen(*)
...
```

libertha.so	libberthaserial.so	libberthaparalleshm.so

PyBERTHA a Python binding for BERTHA

```
#ifdef USEINTELCMP
...
#define f_bertha_main bertha_wrapper_mp_bertha_main_
...
#else
...
#define f_bertha_main __bertha_wrapper_MOD_bertha_main
...
#endif
...
void f_bertha_main (char *, char *, char *, char *,  double *, double *,
    double *, double *, int, int, int, int);
...
```

```
...
int mainrun(char * fittcoefffname, char * vctfilename,
    char * ovapfilename, char * fittfname, double * eigen,
    double * ovapin, double * eigenv, double * fockin)
{
  f_bertha_main(fittcoefffname, vctfilename,
        ovapfilename, fittfname, eigen, ovapin, eigenv, fockin,
        strlen(fittcoefffname), strlen(vctfilename), strlen(ovapfilename),
        strlen(fittfname));
...
```

PyBERTHA a Python binding for BERTHA

```
...
class pybertha:

    def __init__(self, sopath="./bertha_wrapper.so"):
        """
        param: sopath is needed to specify the
        bertha_wrapper Shared Object file.
        """
        soname = sopath
        if (not os.path.isfile(soname) ):
            raise Error("SO %s  does not exist" % soname)

        self.__bertha = ctypes.cdll.LoadLibrary(soname)

        self.__reset()
...
```

PyBERTHA a Python binding for BERTHA

```
...
    def run(self):
        """
        This is the method to perform the SCF computation.
        """

        if self.__init:
            ndim = self.get_ndim()

            eigen = numpy.zeros(ndim, dtype=numpy.double)
            eigen = numpy.ascontiguousarray(eigen, dtype=numpy.double)

           ....
```

PyBERTHA a Python binding for BERTHA

```
...
        main= threading.Thread(target=self.__bertha.mainrun, \
            args=[in_fittcoefffname, \
                in_vctfilename, \
                in_ovapfilename, \
                in_fittfname, \
                ctypes.c_void_p(eigen.ctypes.data), \
                ctypes.c_void_p(ovapbuffer.ctypes.data), \
                ctypes.c_void_p(eigenvctbu.ctypes.data), \
                ctypes.c_void_p(fockbuffer.ctypes.data)])
        maint.daemon = True
        maint.start()
        while maint.is_alive():
            maint.join(.1)
        eigem = doublevct_to_complexmat (eigenvctbu, ndim)
        if eigem is None:
            raise Error("Error in ovap matrix size"
```

PyBERTHA a Python binding for BERTHA

```
def complexmat_to_doublevct (inm):

    if len(inm.shape) != 2:
        return None

    if inm.shape[0] != inm.shape[1]:
        return None

    dim = inm.shape[0]

    cbuffer = numpy.zeros((2*dim*dim), dtype=numpy.double)
    cbuffer = numpy.ascontiguousarray(cbuffer, dtype=numpy.double)

    cbuffer[0::2] = inm.flatten().real
    cbuffer[1::2] = inm.flatten().imag

    return cbuffer
```

PyBERTHA a Python binding for BERTHA

```
def doublevct_to_complexmat (invector, dim):

    if (invector.size != (2*dim*dim)):
        return None

    outm = numpy.zeros((dim,dim), dtype=numpy.complex128)

    inmtxreal = numpy.reshape(invector[0::2], (dim,dim))
    inmtximag = numpy.reshape(invector[1::2], (dim,dim))
    outm[:,:] = inmtxreal[:,:] + 1j * inmtximag[:,:]

    return outm
```

PyBERTHA a Python binding for BERTHA

Algorithm 2 A simple four-component relativistic DFT program implemented using the *berthamod* Python module

```
import berthamod
Inputs: input options ...
bertha = berthamod.pybertha(wrapperso)
bertha.set_verbosity(verbosity)
bertha.set_fnameinput(inputfilename)
bertha.set_fittfname(fittfilename)
bertha.set_tresh(tresh)
bertha.init()
ovapmtx, eigenvectors, fockmtx, eigenvalues = bertha.run()
etotal = bertha.get_etotal()
bertha.finalize()
Output: Total Energy and MO energies ...
```

PyBERTHA a Python binding for BERTHA

Impact of the Python binding in the total execution time using 10 SCF iterations. The code has been executed on a Intel(R) Xeon(R) CPU E3- 1220 compiling the code with the Intel(R) compiler version: 2018.3.222

System	Matrix Dimension	Wall-time 10 SCF iterations with Python (s)	Wall-time 10 SCF iterations without Python (s)	Python overhead 10 SCF iterations
H_2O	140	3.910	3.906	0.09 %
Zn	624	20.471	20.455	0.07 %
Cd	916	41.595	41.556	0.09 %
Hg	1240	97.046	96.975	0.07 %
Au_2	1560	104.458	104.354	0.99 %
Au_4	3152	613.912	613.483	0.07 %
Au_8	6304	3965.911	3964.078	0.05 %

PyBERTHA a Python binding for BERTHA

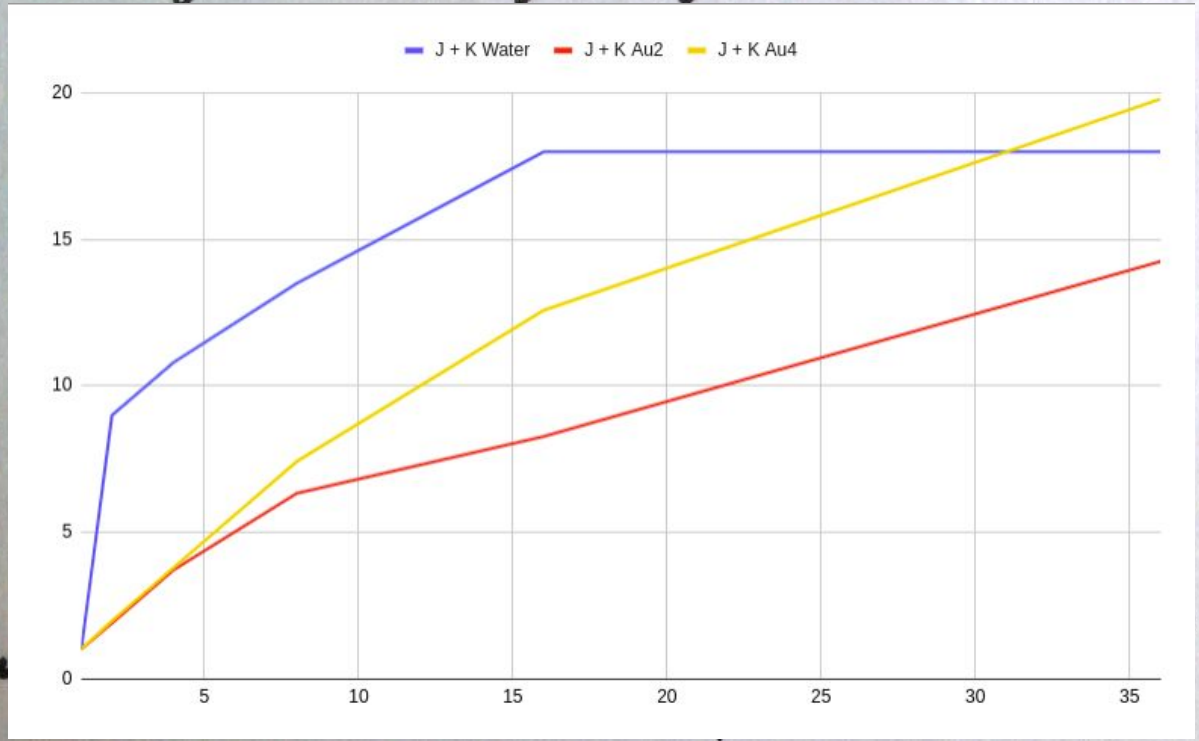
Impact of the Python binding in the berthamod.get_realtime_fock method.

System	Matrix Dimension	Wall-time 10 SCF iterations with Python (s)	Wall-time 10 SCF iterations without Python (s)	Python overhead 10 SCF iterations
H_2O	140	0.383	0.382	0.19 %
Zn	624	1.257	1.250	0.52 %
Cd	916	2.592	2.575	0.64 %
Hg	1240	6.388	6.358	0.48 %
Au_2	1560	6.395	6.294	1.59 %
Au_4	3152	38.050	37.479	1.50 %
Au_8	6304	244.447	241.999	1.00 %

PyBERTHA a Python binding for BERTHA

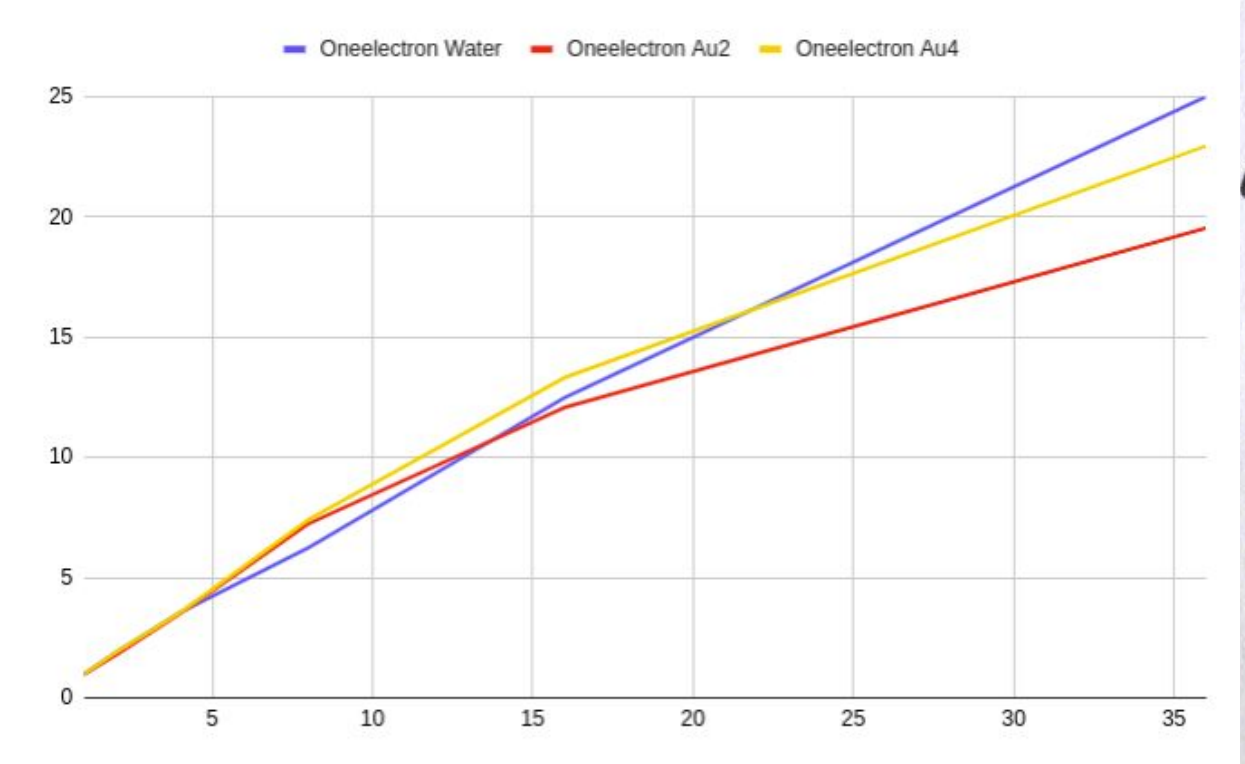
- Parallel PyBERTHA
  - First option load the liberrthaparallelshm.so and after we can manage MPI at python level using mpi4py
  - An other option is to use OpenMP, especially when one is interested in improving the performances for small molecular systems

PyBERTHA a Python binding for BERTHA



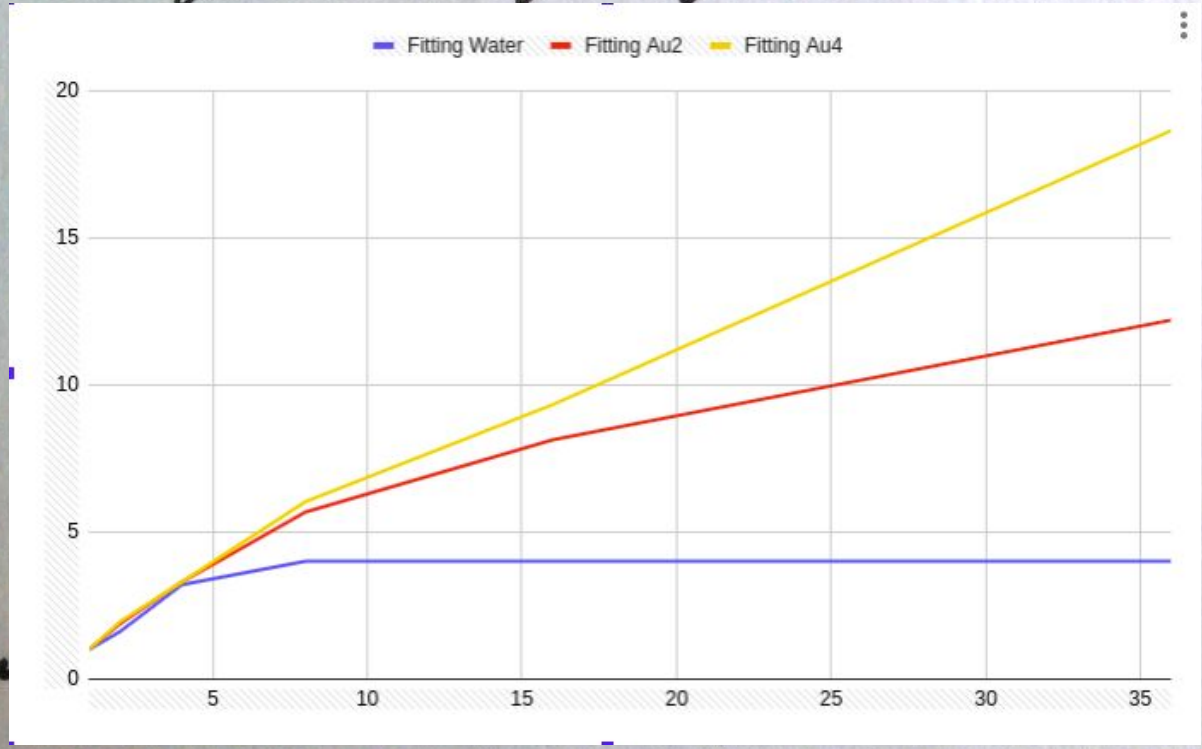
OpenMP palatalization run on Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

PyBERTHA a Python binding for BERTHA



OpenMP palatalization run on Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

PyBERTHA a Python binding for BERTHA



OpenMP palatalization run on Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz

t's try it anyway...

$$(i\gamma^\mu \partial_\mu - m)\psi = 0$$

Examples and applications

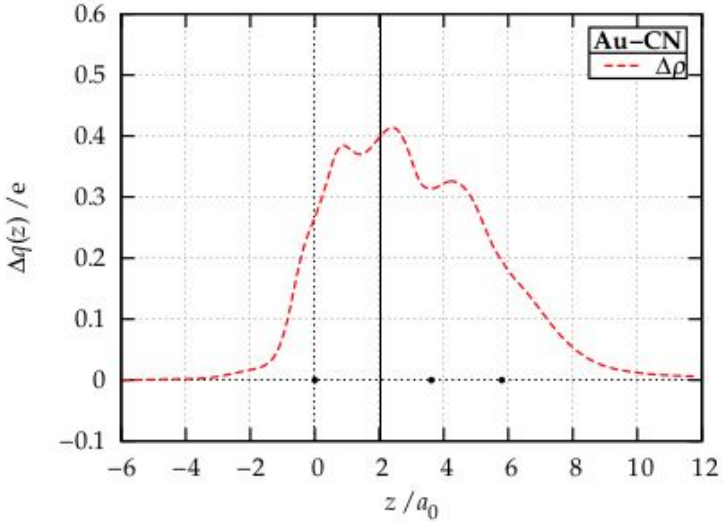
mation, want this, but $i\gamma^\mu$ not four

A test application using NOCV/CD

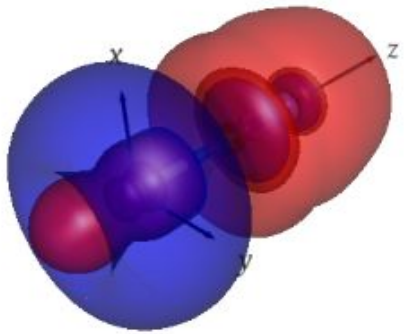
- CD: Charge-Displacement analysis has been successfully employed to describe the nature of intermolecular interactions and various type of controversial chemical bond
- Charge-Displacement function defined as a partial integration along a suitable z axis of the difference $\Delta\rho(x, y, z')$ between the electron density of the adduct and that of its non-interacting fragments placed at the same equilibrium position they occupy in the adduct.

$$\Delta q(z) = \int_{-\infty}^{z} dz' \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \Delta\rho(x, y, z') dx dy$$

A test application using NOCV/CD



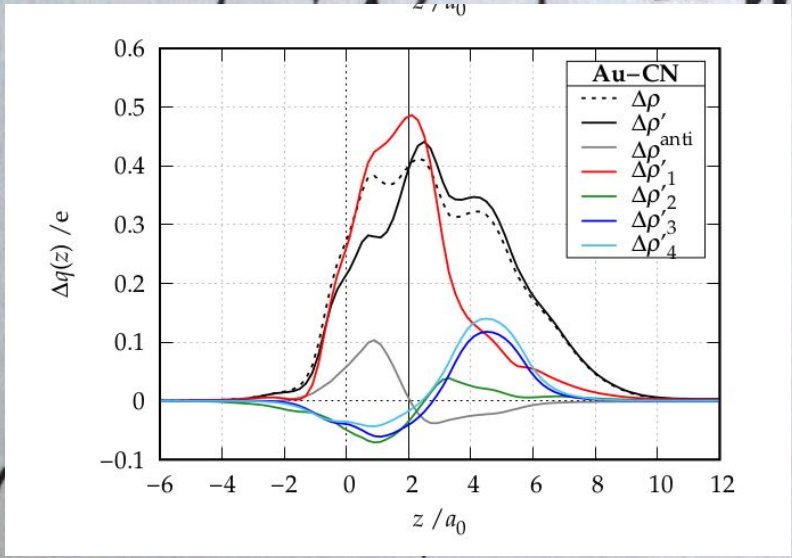
(a) Charge-Displacement function for Au-CN

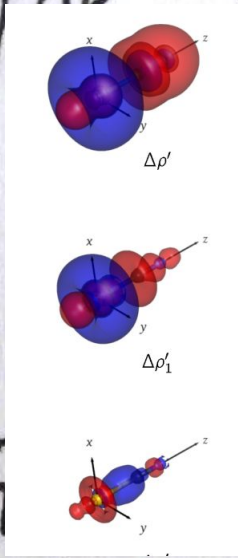


(b) $\Delta\rho = \rho^{(AB)} - \rho^{(A)} - \rho^{(B)}$

A test application using NOCV/CD

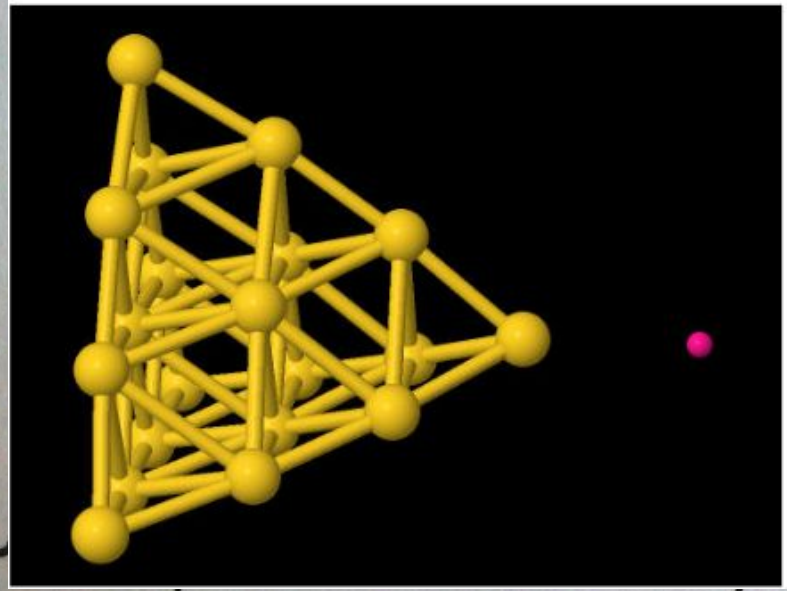
- The core idea of the approach is the decomposition, via natural orbitals for chemical valence (NOCV), of the so-called charge-displacement (CD) function into additive Chemically meaningful components.

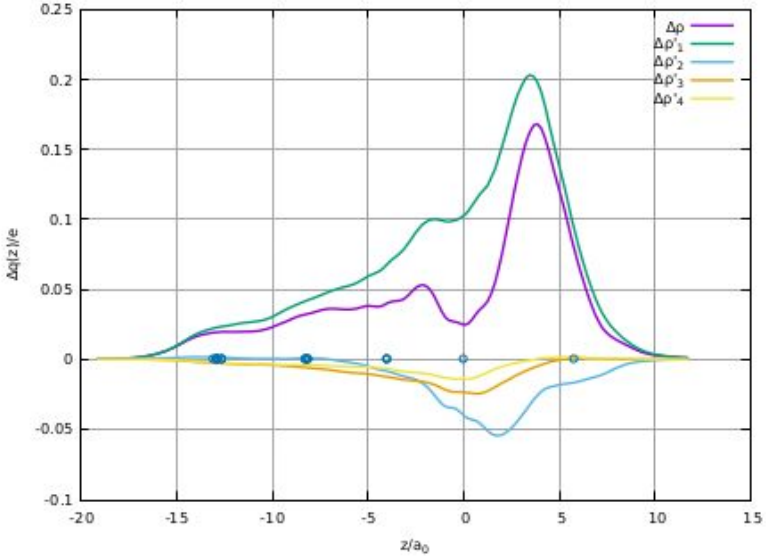




A test application using NOCV/CD

The Au_{20} - FI complex and the CD analysis for the bond





PyBERTHART a real-time TDDFT implementation

The time-dependent equation for Hartee-Fock and density functional theory can be reformulated in terms of the Liouville-von Neumann (LvN) equation. In an orthonormal basis set the LvN equation reads:

$$\imath \frac{\partial \boldsymbol{D}(t)}{\partial t} = \boldsymbol{F}(t)\boldsymbol{D}(t) - \boldsymbol{D}(t)\boldsymbol{F}(t)$$

D(t) and F (t) are the one-electron density matrix and time-dependent Fock matrix respectively

$$\boldsymbol{F}(t) = \boldsymbol{H}_{core} + \boldsymbol{J}[\boldsymbol{D}(t)] + \boldsymbol{V}^{XC}[\boldsymbol{D}(t)] + \boldsymbol{V}^{ext}(t)$$

PyBERTHART a real-time TDDFT implementation

The solution we are seeking for the density matrix reads as:

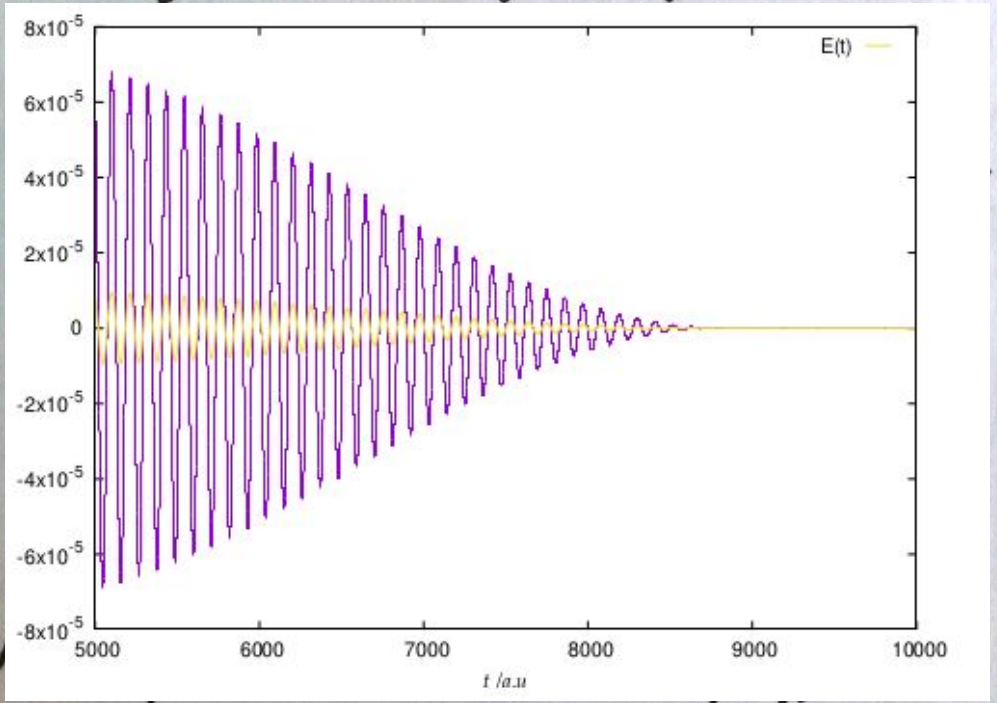
$$\boldsymbol{D}(t) = \boldsymbol{U}(t, t_0)\, \boldsymbol{D}(t_0)\, \boldsymbol{U}(t, t_0)^{\dagger}$$

Within a small time step Δt, the time evolution operator U is defined according to the exponential midpoint rule as:

$$\boldsymbol{U}(t + \Delta t, t) = exp\{-i\boldsymbol{F}(t + \Delta t/2)\Delta t\}$$

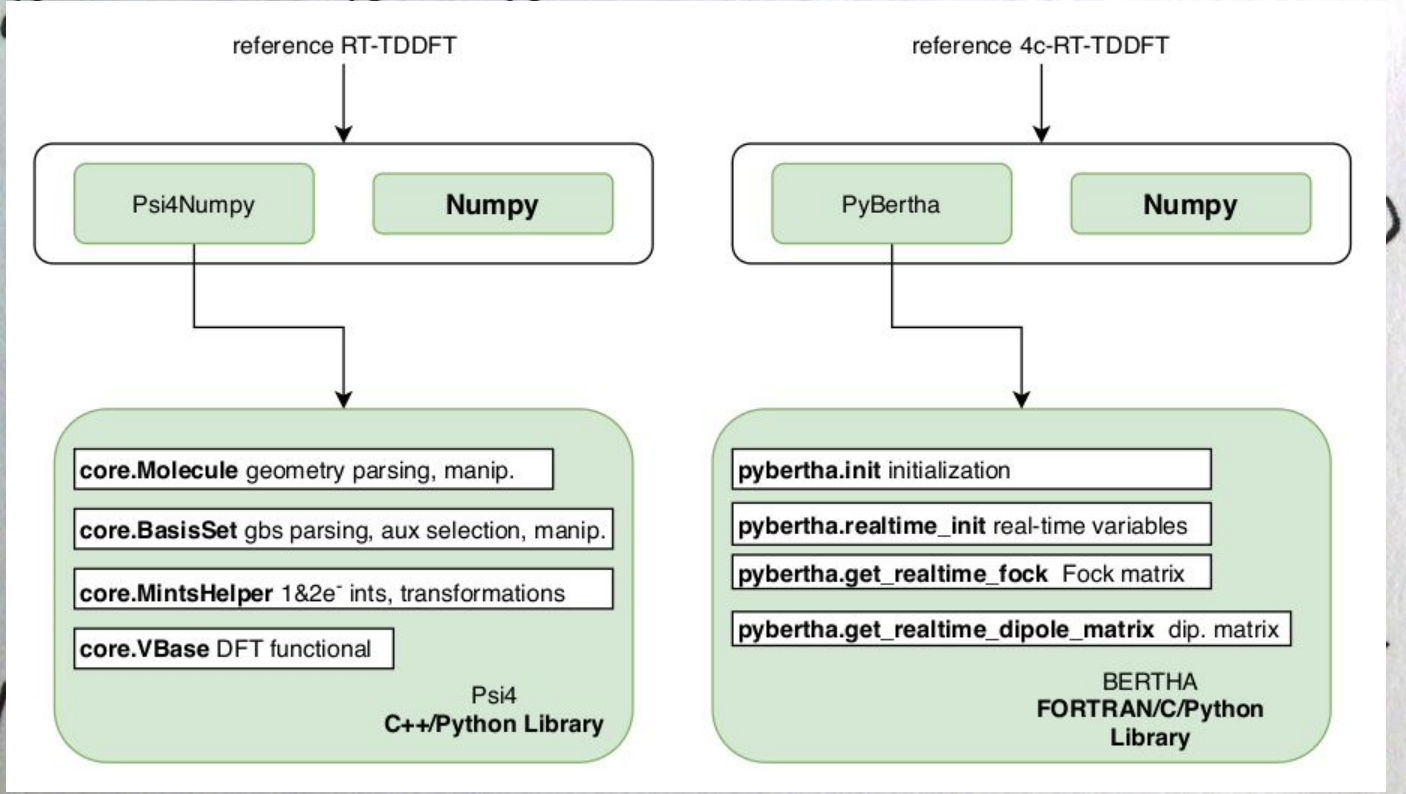
A finite time propagation is carried out by repeatedly applying thepropagator in each time step.

PyBERTHART a real-time TDDFT implementation

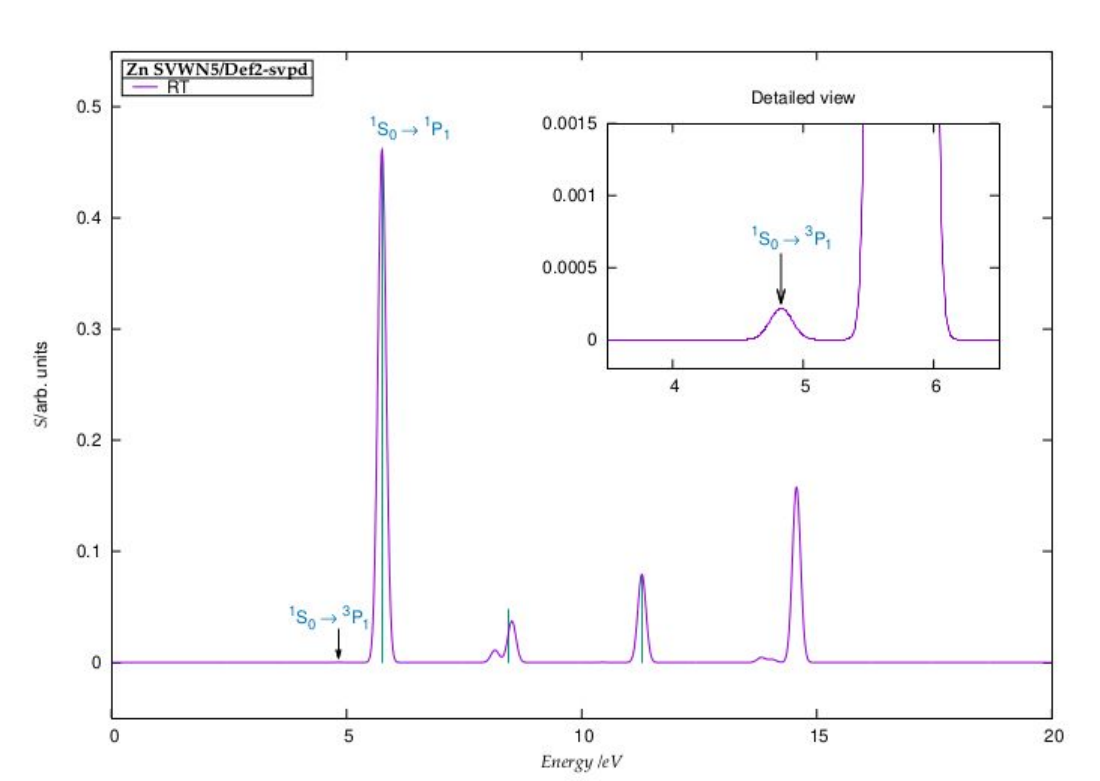


Induced dipole moment in H_2 molecule. The representation of the external field is also reported as a yellow line.

PyBERTHART a real-time TDDFT implementation



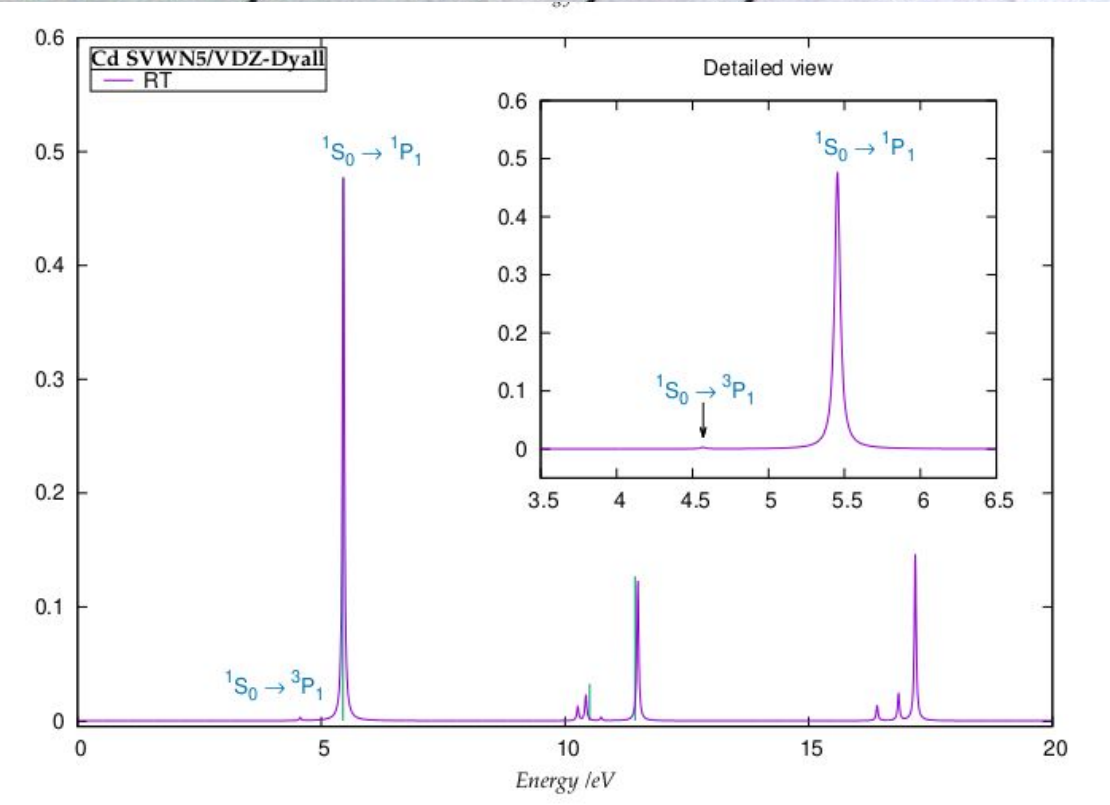
PyBERTHART a real-time TDDFT implementation



The absorption spectrum of group 12 atoms (Zn,Cd,Hg) features spin-forbidden transitions.

A proper relativistic framework is needed in order to reproduce forbidden transitions

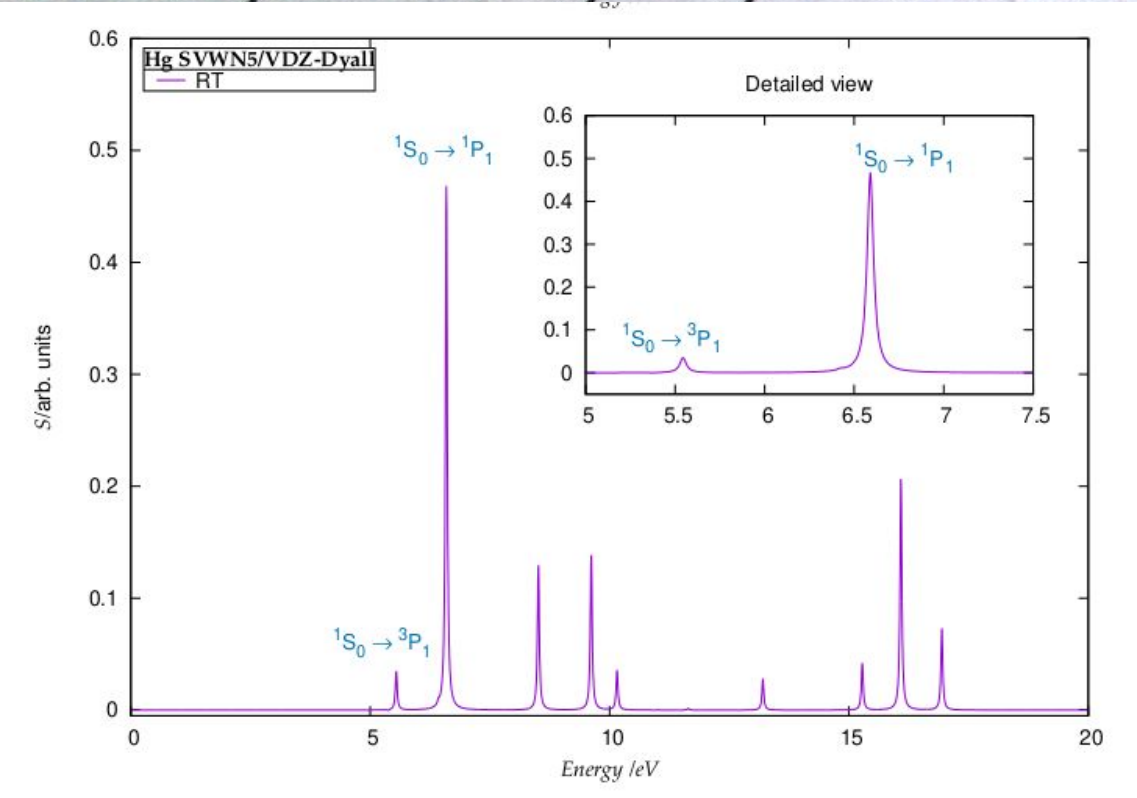
PyBERTHART a real-time TDDFT implementation



The absorption spectrum of group 12 atoms (Zn,Cd,Hg) features spin-forbidden transitions.

A proper relativistic framework is needed in order to reproduce forbidden transitions

PyBERTHART a real-time TDDFT implementation



The absorption spectrum of group 12 atoms (Zn,Cd,Hg) features spin-forbidden transitions.

A proper relativistic framework is needed in order to reproduce forbidden transitions

t's try it anyway...

$$(i\gamma^\mu \partial_\mu - m)\psi = 0$$

Conclusions

mehow, want this, but $i\gamma^\mu$ not four

Conclusions

- We improved and extended the usability of the software both in term of user experience (PyBERTHA) and in terms of molecular system dimensions the user ia able to deal with (Parallelization)
- Full parallel implementation of the DKS module of the program BERTHA featuring (i) no serial portions of code and (ii) a complete distributed memory approach. Indeed, all-electron four-component DKS calculations on systems as costly as the Au_{32} gold cluster, with more than 25 000 basis functions are now feasible with BERTHA provided that a minimal amount of memory per core (as small as 2 GiB) is available
- It is now easy to extend the API as needed. In a future coming version we are planning to add all the fundamental functions to specify the input geometry and basis set in a more user-friendly (i.e. pythonic) way.

REHE2020

13-th International Conference on Relativistic Effects in Heavy-Element Chemistry and Physics

https://www.rehe2020.it/