ELSI Interface (May 2017)
User's Guide

William P. Huhn and Victor Wen-zhe Yu
http://elsi-interchange.org

August 18, 2017

Contents

1 Introduction

1.1 The Cubic Wall of Kohn-Sham Density-Functional Theory

1.1.1 The Kohn-Sham Eigenvalue Problem

In Kohn-Sham density-functional theory (KS-DFT)[1], the multi-reference many-electron wavefunction is rewritten as a Slater determinant of single-particle eigenstates via the Kohn-Sham ansatz. The single-particle eigenstates are the solutions of a set of single-particle eigenvalue problems known as the Kohn-Sham equations:

$$\hat{h}^{\mathrm{KS}}[n]\psi_l = \epsilon_l \psi_l \tag{1.1}$$

where l is an index used to label the eigenvalue problem (in practice, the good quantum numbers of the system), n is the electron density, ψ_l and ϵ_l are Kohn-Sham orbitals and their associated single-particle energies, and $\hat{h}^{\mathrm{KS}}$ is the effective scalar-relativistic Kohn-Sham Hamiltonian:

$$\hat{h}^{\mathrm{KS}}[n] = \hat{t}_{\mathrm{sr}} + \hat{v}_{\mathrm{es}} + \hat{v}_{\mathrm{xc}} + \hat{v}_{\mathrm{ext}} \tag{1.2}$$

which includes the scalar-relativistic kinetic energy $\hat{t}_{\mathrm{sr}}$, the average electrostatic or Hartree potential $\hat{v}_{\mathrm{es}}$ of the electron density and of the nuclei, the exchange-correlation potential $\hat{v}_{\mathrm{xc}}$, and the potential $\hat{v}_{\mathrm{ext}}$ arising from external electromagnetic fields. Eq. 1.1 defines an infinite set of equations; however, there is a lowest bound to the eigenvalues ϵ_l, and most applications in density-functional theory require only a subset of eigenstates with the lowest energy eigenvalues.

For most density-functional codes, a finite set of basis functions are employed to expand the Kohn-Sham orbitals:

$$\psi_l(\boldsymbol{r}) = \sum_{j=1}^{N_{\mathrm{basis}}} c_{jl}\phi_j(\boldsymbol{r}) \tag{1.3}$$

where N_{basis} is the number of basis functions in the set. The choice of basis set determines much of the implementation details of an electronic structure code. When using this discretization, at most N_{basis}-many Kohn-Sham eigenstates may be uniquely determined.

When the Kohn-Sham eigenvectors are discretized in terms of a basis set, each Kohn-Sham eigenvalue problem has the form

$$\sum_{j=1}^{N_{\mathrm{basis}}} h_{ij}c_{jl} = \epsilon_l \sum_{j=1}^{N_{\mathrm{basis}}} s_{ij}c_{jl}, \tag{1.4}$$

where

$$h_{ij} = \int d^3r[\phi_i^*(\boldsymbol{r})\hat{h}^{\mathrm{KS}}\phi_j(\boldsymbol{r})] \tag{1.5}$$

are known as the Hamiltonian matrix elements and

$$s_{ij} = \int d^3r[\phi_i^*(\boldsymbol{r})\phi_j(\boldsymbol{r})] \tag{1.6}$$

are known as the overlap matrix elements. The calculation of the Hamiltonian and overlap matrix elements are implementation-specific, but Eq. 1.4 holds across all KS-DFT codes.

The set of single-particle equations in Eq. 1.4 may be compactly expressed as a single equation using a matrix form,

$$\boldsymbol{H}\boldsymbol{c} = \boldsymbol{\epsilon}\boldsymbol{S}\boldsymbol{c} \tag{1.7}$$

where all matrices have dimensions $N_{\text{basis}} \times N_{\text{basis}}$. The matrix $\boldsymbol{c}$ and diagonal matrix $\boldsymbol{\epsilon}$ contain the eigenvectors and eigenvalues of the (generalized) eigensystem of $\boldsymbol{H}$ and $\boldsymbol{S}$ matrices. In the special case of orthonormal basis sets, the eigenvalue problem described in Eq. 1.7 reduces to the standard form

$$\boldsymbol{H}\boldsymbol{c} = \boldsymbol{\epsilon}\boldsymbol{c}. \tag{1.8}$$

In the more general case of non-orthogonal basis functions (e.g. Gaussian functions, Slater functions, numeric atom-centered orbitals), Eq. 1.7 must be solved.

1.1.2 The Electron Density

The target quantity to be optimized in KS-DFT is the electron density

$$n(\boldsymbol{r}) = \sum_{l=1}^{N_{\text{basis}}} f_l \psi_l^*(\boldsymbol{r})\psi_l(\boldsymbol{r}). \tag{1.9}$$

where f_l is the occupation number for the jth orbital. Optimization of the electron density preferentially selects the eigenstates with the lowest energy eigenvalues, justifying the usage of discretization in Eq. 1.3 for most applications in density-functional theory.

Eq. 1.9 is a textbook equation only suited for small systems due to an unfavorable $O(N_{\text{basis}}^2)$ scaling. The electron density is usually re-expressed in electronic structure theory in a density-matrix form

$$n(\boldsymbol{r}) = \sum_{i,j=1}^{N_{\text{basis}}} \phi_i^*(\boldsymbol{r}) n_{ij} \phi_j(\boldsymbol{r}). \tag{1.10}$$

which has an $O(N_{\text{basis}})$ solution for localized basis elements. Eq. 1.10 may be shown to be equivalent to Eq. 1.9 via

$$n_{ij} = \sum_{l=1}^{N_{\text{basis}}} f_l c_{il} c_{jl}. \tag{1.11}$$

Calculating the density matrix elements via Eq. 1.11 is straightforward, but the eigenvalues and eigenvectors of the Kohn-Sham eigenvalue equations must be known. This requires a direct solution of Eq. 1.7 or 1.8 by an eigensolver, for which only $O(N_{\text{basis}}^3)$ implementations are known. It is this "cubic wall of KS-DFT" that is the the current bottleneck impeding large-scale Kohn-Sham density-functional theory.

For a standard semi-local KS-DFT problem, the eigenvectors of the system *only* enter into the self-consistent cycle via Eq. 1.11. Alternative theoretical approaches, such as the density purification method and the orbital minimization method, generate n_{ij} directly from $\boldsymbol{H}$ and $\boldsymbol{S}$ and bypass the computationally expensive $O(N_{\text{basis}}^3)$ eigenvalue problem. These alternative theoretical approaches often have reduced scaling coefficients, and in some limited cases are expected to approach $O(N_{\text{basis}})$ scaling, the holy grail of large-scale density-functional theory.

1.1.3 Practical Considerations

There is no best solution to the cubic wall problem. While density-matrix-based solvers such as libOMM[2] may have an performance edge over eigensolvers such as ELPA[3, 4] on the semi-local level of theory, they are ill-suited for higher levels of theory (hybrid-functional and many-body perturbative methods) where the eigenvectors of the system are explicitly required. Methods such as PEXSI[5, 6] which provide more favorable scaling for large-scale physical systems often have significant overhead that hinders performance for small and medium-sized systems. Acceleration of the solution (or circumvention) of the Kohn-Sham eigenvalue problems requires multiple electronic structure solvers be interfaced with a KS-DFT code.

Another important implementation detail is the matrix storage format. Multiple matrix storage formats, both dense and sparse, are used in the wild. A non-comprehensive list includes upper/lower triangular, compressed row/column, cyclic distribution, and block distribution in 1-dimension (matrix row or column) or 2-dimension (both row and column). Interfacing a electronic structure solver with a KS-DFT code will usually require some form of matrix conversion wrapper, costing valuable development and validation time.

1.2 ELSI, the ELectronic Structure Infrastructure

ELSI unifies the community effort in overcoming the cubic-wall problem of KS-DFT by bridging the divide between developers of electronic structure solvers and KS-DFT codes. ELSI gives KS-DFT developers access to multiple solvers via a unified interface, with matrix format conversion is done automatically by ELSI. Solvers are treated on equal footing within ELSI, giving solver developers a unified platform for implementation and benchmarking across codes and physical systems. Solvers can be switched dynamically in the middle of an SCF cycle, allowing the KS-DFT developer to mix-and-match strengths of different solvers. Solvers can work cooperatively with one another within ELSI, allowing for acceleration greater than either solver can achieve individually. In the future, we will implement a decision layer that automatically determines the solver most suited for a problem based on the attributes of the problem (matrix size, sparsity, etc.) Most importantly, ELSI exists as a community for KS-DFT and solver developers to interact and work together to improve performance of solvers, with monthly web meetings to discuss progress on code development, yearly on-site connectors meetings, and planned webinars and workshops.

In the ELSI May 2017 release, ELPA[3, 4], libOMM[2], and PEXSI[5, 6] are supported by ELSI. Work is undergoing to provide external linkage with CheSS[7] and SIPs[8] solvers. Codes currently integrated with ELSI include DGDFT[9], FHI-aims[10], NWChem[11] via Global Arrays, and SIESTA[12], with commitments for future integration by BigDFT[13].

1.2.1 Design Tenets of ELSI

Versatility: ELSI supports density matrix and eigensystem formalisms on equal footing. It enables calculation of eigenvalues and eigenvectors (real-valued and complex-valued) via ELPA and calculation of the density matrix (real-valued) via ELPA, libOMM, or PEXSI. A unified software interface designed for rapid integration into a variety of KS-DFT codes is provided. Fortran and C/C++ interfaces are provided, and a Python interface in progress.

Flexibility: ELSI supports both dense and sparse matrices as input/output formats. The default dense matrix format is the "2D block-cyclic distributed dense matrix" as used in BLACS/ScaLAPACK[14]; the default sparse matrix format is the "1D block distributed compressed sparse column matrix" as used in PEXSI[15]. In situations where the input/output matrix storage format requested by the calling KS-DFT code and the matrix storage format used internally by the requested solver are different, conversion and redistribution of matrices will be performed automatically.

Scalability: The solver libraries collected in ELSI are highly scalable. ELPA can scale to ∼100k CPU cores given a sufficiently large problem to solve, and PEXSI, with its efficient two-level parallelism, easily scales to ∼500k CPU cores.

Portability: Whenever possible, ELSI redistributes source packages for supported solvers, which the user may build alongside the ELSI interface layer via ELSI's unified make system. ELSI and its redistributed library source packages have been confirmed to work on commonly-used HPC architectures (Cray, IBM, Intel, NVIDIA) using all major compilers (GNU, Intel, IBM, PGI, Cray). ELSI also provides the option for external linkage against preinstalled solver libraries for all supported solvers.

Extensibility: As the ELSI Interface software evolves, more features will be available with minimal API changes. Currently ongoing developments include (1) supporting complex-valued density matrix calculation, (2) optimizing the performance of ELPA on hybrid CPU + GPU architectures (e.g. POWER) and on multi-core architectures (e.g. Intel Knights Landing), and (3) integrating the SIPs sparse eigensolver[8] and CheSS sparse density matrix solver[7] into ELSI Interface.

1.3 Solver Libraries Supported by ELSI

1.3.1 ELPA: Eigenvalue soLvers for Petaflop-Applications

The Kohn-Sham eigenvalue problem (1.4) can be explicitly solved by traditional (tri)diagonalization. The massively parallel direct solver ELPA[4, 3] facilitates the solution of symmetric or Hermitian eigenvalue problems on high-performance computers. It was initially designed for distributed memory architectures and later extended to exploit multi-threading parallelism. GPU acceleration is currently under development.

In ELPA, the generalized eigenvalue problem (1.4) is first transformed into the standard form by using Cholesky factorization of the overlap matrix $\boldsymbol{S}$:

$$\boldsymbol{S} = \boldsymbol{L}\boldsymbol{L}^* \tag{1.12}$$

where $\boldsymbol{L}$ is a lower triangular matrix. The Cholesky factorization allows the eigenproblem to be efficiently transformed to a standardized form

$$\tilde{\boldsymbol{H}}\tilde{\boldsymbol{c}} = \boldsymbol{\lambda}\tilde{\boldsymbol{c}}, \tag{1.13}$$

where $\tilde{\boldsymbol{H}} = \boldsymbol{L}^{-1}\boldsymbol{H}(\boldsymbol{L}^*)^{-1}$ and $\tilde{\boldsymbol{c}} = \boldsymbol{L}^*\boldsymbol{c}$.

The standard eigenvalue problem is subsequently reduced further, either directly to the tridiagonal form:

$$\boldsymbol{T} = \boldsymbol{Q}\tilde{\boldsymbol{H}}\boldsymbol{Q}^* \tag{1.14}$$

or first reduced to an intermediate banded form

$$\boldsymbol{B} = \boldsymbol{Q}_1\tilde{\boldsymbol{H}}\boldsymbol{Q}_1^* \tag{1.15}$$

followed by the tridiagonal form:

$$\boldsymbol{T} = \boldsymbol{Q}_2\boldsymbol{B}\boldsymbol{Q}_2^* \tag{1.16}$$

In Eqs. 1.14, 1.15, 1.16, $\boldsymbol{Q}$, $\boldsymbol{Q}_1$, $\boldsymbol{Q}_2$ are transformation matrices, $\boldsymbol{T}$ is a tridiagonal matrix, and $\boldsymbol{B}$ is a banded matrix. The first reduction strategy is known as "one-stage" tridiagonalization, and the second strategy is known as "two-stage" tridiagonalization.

The key steps of the two-stage tridiagonalization algorithm, as implemented in ELPA, are illustrated in Fig. 1.1. Although the two-stage tridiagonalization has two additional steps (steps 2 and 4), it has been demonstrated to be more efficient and exhibit better scalability compared with the one-stage approach, as many matrix-vector operations (BLAS level-2 routines) in the one-stage tridiagonalization can be replaced by more efficient matrix-matrix operations (BLAS level-3 routines).

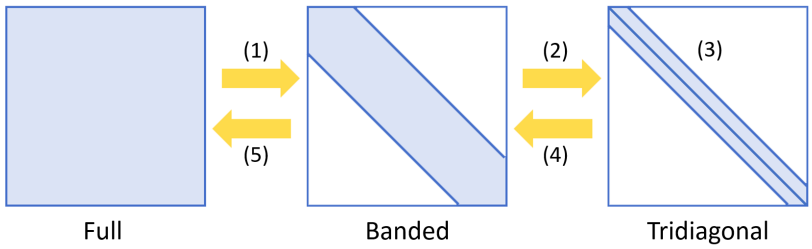

Figure 1.1: Five computational steps of the ELPA eigensolver with two-stage tridiagonalization. (1) Reduction from the full matrix to a banded form. (2) Reduction from the banded matrix to a tridiagonal form. (3) Solve the eigenvalues and eigenvectors of the tridiagonal system. (4) Back-transform the eigenvectors to the banded form. (5) Back-transform the eigenvectors to the original full form.

The eigenvalues and eigenvectors of the tridiagonal eigenproblem can be efficiently computed by a divide-and-conquer algorithm. The eigenvectors of the tridiagonal system must be subsequently back-transformed into the standardized

eigenproblem form. However, in KS-DFT calculations, often only a small fraction of the eigenvectors needs to be back-transformed, in which case the two-stage tridiagonalization is known to give the best performance in terms of speed and scalability.

Since ELPA employs the same 2D block-cyclic matrix distribution as ScaLAPACK (BLACS), it can easily be substituted into existing codes that already support ScaLAPACK.

1.3.2 libOMM: Orbital Minimization Method

The orbital minimization method (OMM) obtains the density matrix directly by using efficient iterative algorithms to minimize an unconstrained energy functional, bypassing the diagonalization of the $N_{\text{basis}} \times N_{\text{basis}}$ eigenproblem required by Equation 1.11. This energy functional is defined in terms of a set of auxiliary orbitals which are *not* the Kohn-Sham orbitals ϕ_i. Rather, the OMM employs a set of non-orthogonal Wannier functions

$$\chi_k = \sum_{j=1}^{N_{\text{basis}}} W_{kj}\phi_j \tag{1.17}$$

to represent the occupied subspace of the system. The number of Wannier functions that OMM uses is $N_{\text{W}} = N_{\text{electron}}/2$, where N_{electron} is the number of electrons in the system.

The $\boldsymbol{H}$ and $\boldsymbol{S}$ matrices projected onto the reduced subspace of Wannier functions are

$$\boldsymbol{H}_{\text{omm}} = \boldsymbol{W}^* \boldsymbol{H} \boldsymbol{W} \tag{1.18}$$

and

$$\boldsymbol{S}_{\text{omm}} = \boldsymbol{W}^* \boldsymbol{S} \boldsymbol{W} \tag{1.19}$$

where $\boldsymbol{W}$ is the coefficients matrix of the Wannier functions. The size change of the Hamiltonian matrix facilitated by Eq. (1.18) is showed in Fig. 1.2.

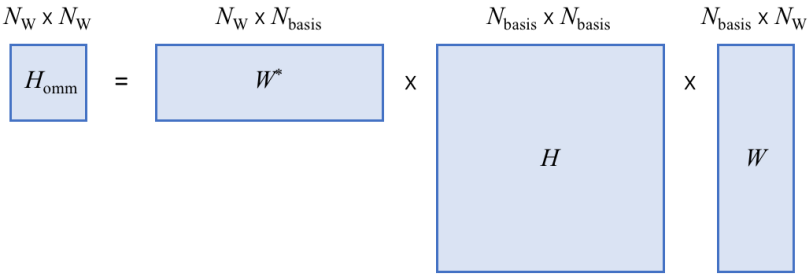

Figure 1.2: Sizes of Hamiltonian matrix before and after applying the Wannier function transformation in the orbital minimization method. Matrix dimensions are shown above the matrices. N_{W}: Number of Wannier functions. N_{basis}: Number of basis functions.

The OMM energy functional defined on the reduced subspace is

$$E[\boldsymbol{W}] = 4Tr[\boldsymbol{H}_{\text{omm}}] - 2Tr[\boldsymbol{S}_{\text{omm}}\boldsymbol{H}_{\text{omm}}] \tag{1.20}$$

This functional, when minimized with respect to the coefficients of Wannier functions, can be shown to be equal to the sum of the lowest $N_{\text{electron}}/2$ eigenvalues of the original Kohn-Sham eigenvalue problem.

The orbital minimization method was initially designed as a linear scaling KS-DFT method. Subsequent implementations in libOMM, supported by ELSI, are cubic scaling iterative algorithms with minimization performed following a conjugate-gradient (CG) scheme. Although not linear scaling, libOMM exhibits a smaller prefactor compared to direct dense eigensolvers by avoiding diagonalization and working on a reduced subspace. However, since libOMM works exclusively on the occupied subspace of the system, in its current implementation it is formally incapable of treating systems with fractional occupation numbers, e.g. metals.

1.3.3 PEXSI: Pole EXpansion and Selected Inversion

The density matrix in Eq. (1.11) is associated with the Kohn-Sham orbitals and their occupation numbers f_l. The occupation numbers are computed from the Fermi-Dirac distribution function

$$f_l = \frac{1}{1 + e^{\frac{\epsilon_l - \mu}{k_B T}}}, \tag{1.21}$$

where k_B is the Boltzmann constant, T is the temperature, and μ is the chemical potential that is determined by the normalization condition

$$\sum_{l=1}^{N_{\text{basis}}} f_l = N_{\text{electron}}. \tag{1.22}$$

In the Pole EXpansion and Selected Inversion (PEXSI) method, the Fermi-Dirac function is approximated by the linear combination of a small number of rational functions (that is, a pole expansion):

$$n \approx \sum_{l=1}^{P} \Im \left(\omega_l^{\rho} (\boldsymbol{H} - (z_l + \mu)\boldsymbol{S})^{-1} \right). \tag{1.23}$$

The complex shifts $\{z_l\}$ and weights $\{\omega_l^{\rho}\}$ are determined through a semi-analytic formula based on contour integration and take negligible time to compute. The number of terms in the pole expansion is proportional to $\log(\beta \Delta E)$, where $\beta = \frac{1}{k_B T}$ is the inverse of the thermal energy and ΔE is the spectral radius. This logarithmic scaling in number of poles yields a highly efficient approach to expand the Fermi operator. Typically $40 \sim 80$ poles are sufficient for results obtained from PEXSI to be comparable (on the level of μeV/atom) to those obtained from diagonalization.

When targetting the electron density $n(\boldsymbol{r})$, only the entries of $(\boldsymbol{H} - (z_l + \mu)\boldsymbol{S})^{-1}$ corresponding to the non-zero pattern of $\boldsymbol{H}$ and $\boldsymbol{S}$ are needed. A selected inversion algorithm can be used to efficiently compute these selected elements, and thus the electron density. The formal computational complexity of PEXSI is $O(N)$ for quasi-1D systems, $O(N^{1.5})$ for quasi-2D systems, and $O(N^2)$ for 3D bulk systems. This favorable scaling does not rely on the gap of the system, i.e. PEXSI supports metals as well as insulators and semiconductors.

The PEXSI method has a two-level parallelism structure and is highly scalable by design. A recently-developed massively-parallel implementation of PEXSI has been shown to exhibit favorable scaling using $10,000 \sim 100,000$ MPI tasks on high performance machines.

1.4 The ELSI Team

Members of the ELSI team that contributed to ELSI's git repository are (ordered by number of commits:)

Victor Yu (website)	Duke University
Björn Lange	Duke University; now DACS Laboratories GmbH
William Huhn (website)	Duke University
Álvaro Vázquez-Mayagoitia (website)	Argonne National Laboratory
Wenhui Mi	Duke University
Fabiano Corsetti (website)	Imperial College London; now QuantumWise
Ali Seifitokaldani	Duke University
Weile Jia	University of California at Berkeley

Additional members of the ELSI team are (ordered alphabetically:)

Volker Blum (website)	Duke University
Alberto Garcia	Institut de Ciència de Materials de Barcelona (ICMAB-CSIC)
Mathias Jacquelin (website)	Lawrence Berkeley National Laboratory
Lin Lin (website)	University of California at Berkeley; Lawrence Berkeley National Laboratory
Jianfeng Lu	Duke University
Chao Yang (website)	Lawrence Berkeley National Laboratory
Haizhao Yang	Duke University

1.5 Acknowledgments

ELSI is a National Science Foundation Software Infrastructure for Sustained Innovation - Scientific Software Integration (SI2-SSI) supported software infrastructure project. The ELSI Interface software and this User's Guide are based upon work supported by the National Science Foundation under Grant Number 1450280. Any opinions, findings, and conclusions or recommendations expressed here are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2 Installation of ELSI

2.1 Overview

The May 2017 release of ELSI contains the ELSI Interface software as well as redistributed source code for the solver libraries ELPA (version 2016.11.001.pre), libOMM (version 0.0.1), and PEXSI (version 0.10.2). Tarballs containing this and future release versions may be downloaded from the ELSI Interchange website at `elsi-interchange.org`. We highly encourage all users to request access to our GitLab server at elsi-team@duke.edu, where we regularly update ELSI between releases while preserving stability.

2.2 Prerequisites

The standard and highly recommended installation of ELSI (compiled with ELPA, libOMM, and PEXSI support) requires the following dependencies:

1. A Fortran 2003 compiler
2. A C compiler supporting the C99 standard
3. A C++ compiler supporting the C++ 11 standard
4. An MPI library
5. LAPACK and ScaLAPACK
6. ParMETIS 4.0.3 (available at http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview)
7. SuperLU_DIST 5.1.2 or 5.1.3 (available at http://crd-legacy.lbl.gov/∼xiaoye/SuperLU)

For those users that are unable to compile ParMETIS or SuperLU_DIST, we also provide a "minimum" installation of ELSI (compiled with ELPA and libOMM support, no PEXSI support) with the following dependencies:

1. A Fortran 2003 compiler
2. A C compiler
3. An MPI library
4. LAPACK and ScaLAPACK

All major compiler suites (GNU, Intel, PGI, XL, and Cray) are supported.

2.3 Installation

The installation of ELSI uses a unified build system which, by default, builds all redistributed solver libraries (ELPA, libOMM, PEXSI) alongside the ELSI interface. Compilation settings are controlled by a make.sys file provided by the user (see Section 2.4.1). If the user already has well-optimized versions of these libraries available, they may link against the individual solvers externally (see Section 2.4.2 for details.)

Once a properly configured make.sys file specifying compilation settings is provided by the user, the installation process of ELSI is a standard make process:

make	Run the unified ELSI build system.
make check	(Optional) Perform a quick test with ELPA, libOMM and PEXSI solvers.
make install	Finalize the ELSI installation.

The resulting libraries, modules, and executables files will be placed in the lib, include, and bin subfolders, respectively, of the ELSI root directory.

The unified ELSI build system will compile all redistributed solvers with the same compilation setting as the ELSI interface layer by default. This behavior may be overwritten by manually specifying compilation flags for each solver (c.f. Appendix A).

Performing regression tests to verify the compiled library by the make check step is highly recommended but not mandatory. Please see Section 2.4.2 for more information about the regression test infrastructure in ELSI.

2.4 Compilation Settings: The make.sys File

The installation of ELSI is controlled a user-specified make.sys file. Example make.sys files can be found in the make_sys subfolder of the root directory, which the user may adapt to their own system architecture. make.sys files for various HPCs are included (Titan, Mira, Theta, Edison, Cori) as well as make.sys files targeted towards standard Linux-based clusters (c.f. the "timewarp" make.sys files.)

2.4.1 Example make.sys File

We provide an example make.sys file, valid for the May 2017 release, using the Intel compiler suite and libraries (Intel MPI, Intel MKL). More details may be found in the next subsection.

```
# PLATFORM  : Generic Cluster running CentOS 6 ("timewarp" @ Duke University)
# COMPILERS : INTEL 14.0
# VERIFIED  : MAY 27, 2017

# ==============================    ELSI    ==================================
#               Compilation settings for the ELSI interface
#          These flags will be used for compiling each redistributed
#         solver unless user explicitly provides flags for said solver
MPIFC         = mpiifort
MPICC         = mpiicc
MPICXX        = mpiicpc
LINKER        = mpiifort

FFLAGS        = -O3
CFLAGS        = -O3
CXXFLAGS      = -O3 -std=c++11
LDFLAGS       = -O3

CXX_LIB       = -lstdc++
SCALAPACK_LIB = -L/PATH/TO/INTEL/MKL/lib/intel64 -lmkl_scalapack_lp64 \
                -lmkl_blacs_intelmpi_lp64 -lmkl_intel_lp64 \
                -lmkl_sequential -lmkl_core -lpthread -lm

# ============================== MAKE CHECK ==================================
#             Flags for running the "make check" part of the ELSI
#             unified make system.  This section is optional but
#             *highly* recommended.
TOMATO_SEED   = /PATH/TO/TOMATO_SEED
MPI_EXEC      = mpirun

# ==============================    ELPA    ==================================
#            Compilation flags specifically targeting ELPA.
#            The Generic kernel is the universal "safe" choice.
#            For optimal ELPA performance, use a better kernel
#            targeted for your system architecture!  See ELPA
#            and/or ELSI documentation for more details.
ELPA2_KERNEL   = Generic

# ==============================    PEXSI   ==================================
#            Compilation flags specifically targeting PEXSI.
METIS_LIB      = -L/PATH/TO/PARMETIS/build/Linux-x86_64/libmetis -lmetis
PARMETIS_LIB   = -L/PATH/TO/PARMETIS/build/Linux-x86_64/libparmetis -lparmetis
SUPERLU_INC    = -I/PATH/TO/SUPERLU_DIST/SRC
SUPERLU_LIB    = -L/PATH/TO/SUPERLU_DIST/lib -lsuperlu_dist_5.1.3
```

2.4.2 Constructing a make.sys File

In this subsection, we list the important keywords that make up a make.sys file. A full list of supported keywords may be found in Appendix A.

Mandatory Keywords

The mandatory set of keywords contained in the make.sys file for a standard ELSI installation are:

MPIFC	= MPI Fortran compiler
MPICC	= MPI C compiler
MPICXX	= MPI C++ compiler
CXX_LIB	= Standard C++ library, e.g. -lstdc++
SCALAPACK_LIB	= LAPACK and ScaLAPACK library flag(s)
METIS_LIB	= METIS library flag(s)
PARMETIS_LIB	= ParMETIS library flag(s)
SUPERLU_INC	= SuperLU_Dist include flag(s)
SUPERLU_LIB	= SuperLU_Dist library flag(s)

These keywords have no default values. The compilation will fail if they are not set properly. However, compiler wrappers used on various supercomputering resources may already include libraries and optimization settings, in which case the associated keywords may be left blank.

The choice of MPI and math libraries make a *huge* difference in the performance of compiled code. Please use the MPI and math libraries recommended by your system administrators.

If the external dependencies ParMETIS and/or SuperLU_DIST cannot be installed by the user, ELSI may be compiled in a "minimum" installation without PEXSI support by including the following keyword in the make.sys file:

DISABLE_CXX	= yes [default: no]

When PEXSI support is disabled, the METIS_LIB, PARMETIS_LIB, SUPERLU_INC, and SUPERLU_LIB keywords may be omitted. The minimum installation should be viewed as a last resort, as PEXSI achieves better performance for large-scale calculations compared to libOMM and ELPA.

Compiler Flags

While the keywords in the previous section are sufficient to compile ELSI, maximizing performance requires suitable optimization flags be set. Below are the keywords associated with compiler flags. It is recommended the user uses optimization flags recommended by system administrators.

FFLAGS	= Fortran compiler flags [default: empty]
CFLAGS	= C compiler flags [default: empty]
CXXFLAGS	= C++ compiler flags [default: empty]
LDFLAGS	= Linker flags [default: $(FFLAGS)]

make check and TOMATO

Regression tests of ELSI are performed using matrices generated by TOMATO (TOy MATrices On-the-fly), a library distributed alongside libOMM in ELSI to rapidly create matrices resembling those encountered in KS-DFT calculations with variable size and sparsity. While the core TOMATO library is distributed with ELSI, template files are required by TOMATO to create matrices resembling realistic systems (polyethylene, boron nitride monolayer, bulk silicon, etc.) These templates are provided as an external package named TOMATO-SEED, stored and maintained as part of the ELSI GitLab.

The path to the TOMATO-SEED folder should be provided by the user using the keyword

TOMATO_SEED = Path to TOMATO-SEED folder [default: empty]

Running the regression tests requires a working MPI wrapper executable and an environment that permits MPI calculations. The MPI wrapper executable may be provided by the user using the keyword:

MPI_EXEC = MPI wrapper executable, e.g. mpiexec, mpirun, aprun, srun [default: mpirun]

We note that many clusters do not allow usage of MPI wrapper executables on nodes where code compilation is permitted. In this case, the user will need to request an interactive job or submit a custom batch script. Please see the cluster's documentation for more information.

ELPA2 Kernels

Solver-specific compilation settings for solver libraries redistributed with ELSI are generally handled by the unified ELSI build system, though they may be manually set by experienced users wishing to override the default behavior (c.f. Appendix A). There is one important exception that all users should set explicitly: the ELPA2 kernel.

The first back-transformation step in ELPA2 for the eigenvectors (c.f. step 4, Section 1.3.1) is a computational bottleneck when a large percentage of all possible eigenvectors are calculated. This step has been heavily optimized using a collection of "kernels" specifically written to take advantage of processor architecture (e.g. vectorization instruction sets). The choice of ELPA2 kernel is specified via the ELPA2_KERNEL keyword in the make.sys file. It is *highly* recommended that the user consults their system administrators and selects the ELPA2 kernel most suited to their system architecture.

Valid choices for the ELPA2_KERNEL keyword are:

Kernel Choice	Description
Generic	The generic ELPA2 Fortran kernel, which is expected to work on all platforms. [default]
BGQ	Fortran code enhanced with assembler calls for IBM Blue Gene/Q
SSE	x86_64 assembler code using SSE2/SSE3 instructions
AVX	Optimized intrinsic code for x86_64 systems using AVX instruction sets
AVX2	Optimized intrinsic code for x86_64 systems using AVX2 instruction sets

External Linkage against Preinstalled Solver Libraries

By default, the unified ELSI build system will build the ELPA, libOMM, and PEXSI solver libraries redistributed with ELSI, as well as the ELSI interface itself. Optionally, the user may link ELSI directly against a preinstalled version of any (or all) supported solver library, bypassing the solver version redistributed with ELSI. It is the responsibility of the user to verify that their preinstalled solver library is properly compiled and that the API of the preinstalled solver library matches the API that ELSI expects from the solver.

The relevant keywords for the make.sys file are below, organized by solver. An example make.sys demonstrating external linkage is provided in ./make_sys/make.sys-external-linkage. The solver version expected by ELSI is also indicated.

1) ELPA 2016.11.001.pre

EXTERNAL_ELPA	= yes [default: no]
ELPA_INC	= ELPA include flag(s) [default: built-in version]
ELPA_LIB	= ELPA library flag(s) [default: built-in version]

2) libOMM 0.0.1

EXTERNAL_OMM	= yes [default: no]
OMM_INC	= libOMM include flag(s) [default: built-in version]
OMM_LIB	= libOMM library flag(s) [default: built-in version]

To enable external libOMM, the four libraries in the OMM bundle (libOMM, MatrixSwitch, PSPBLAS, TOMATO) must be available in OMM_LIB.

3) PEXSI 0.10.2

EXTERNAL_PEXSI	= yes [default: no]
PEXSI_INC	= PEXSI include flag(s) [default: built-in version]
PEXSI_LIB	= PEXSI library flag(s) [default: built-in version]

2.5 Linking and Importing ELSI into KS-DFT Codes

2.5.1 Linking a KS-DFT Code against ELSI

ELSI places its library and include files in the `lib` and `include` subfolders, respectively, of the ELSI root directory. An example set of compiler flags to link a generic KS-DFT code against a standard installation of ELSI are:

```
ELSI_INCLUDE = -I/PATH/TO/ELSI/include
ELSI_LIB     = -L/PATH/TO/ELSI/lib -lelsi \
               -lOMM -ltomato -lMatrixSwitch -lpspblas -lelpa -lpexsi -lcheck_singularity \
               -L$/PATH/TO/SUPERLU/lib -lsuperlu_dist \
               -L$/PATH/TO/PARMETIS/libparmetis -lparmetis \
               -L$/PATH/TO/PARMETIS/libmetis -lmetis
```

These flags will vary based on the user's needs, although the ELSI team has found that this set of flags works well for various KS-DFT codes and compilers. Linking ELSI against preinstalled solver libraries will require the user modify these flags for their environment.

If PEXSI is disabled by setting DISABLE_CXX = yes in the make.sys file (i.e. the "minimum" installation of ELSI), an example set of the resulting compiler flags would be:

```
ELSI_INCLUDE = -I/PATH/TO/ELSI/include
ELSI_LIB     = -L/PATH/TO/ELSI/lib -lelsi \
               -lOMM -ltomato -lMatrixSwitch -lpspblas -lelpa -lcheck_singularity
```

2.5.2 Importing ELSI into a KS-DFT Code

After a KS-DFT code is successfully linked against ELSI, ELSI may be used in the KS-DFT code by importing the appropriate header file. For codes written in Fortran, this is done by using the native ELSI module

```
USE ELSI
```

For codes written in C/C++, the ELSI wrapper around the native Fortran code may be imported by including the ELSI wrapper header file

```
#include <elsi.h>
```

These import statements give the KS-DFT code access to the (public-facing) ELSI interface. In the next chapter, we will describe the API for the ELSI interface.

3 The ELSI API

3.1 Overview of the ELSI API

In this chapter, we present the public-facing API for the interface layer of the ELSI May 2017 release. We anticipate that fine details of this interface (number of arguments in subroutines, encapsulation via types/structs) may change slightly for the July 2017 release, but the fundamental structure of the interface layer is expected to remain consistent. After the July 2017 release, the public-facing interface layer will remain stable for the foreseeable future, and future development will occur internally in ELSI.

ELSI provides a C/C++ wrapper in addition to the native Fortran interface. The ELSI team has endeavored to keep these two interfaces as closely aligned as possible. The vast majority of this chapter, while written from a Fortran-ic standpoint, applies equally to both interfaces. Information specifically about the C/C++ wrapper for ELSI may be found in Section 3.6. A Python wrapper is planned for a future release.

3.1.1 Brief Description of ELSI Subroutines

ELSI has four families of subroutines making up its API: Controllers, Solvers, Customizers, and Utilities. Controllers are used to set up required parameters for ELSI and extract additional quantities from solvers. Solvers are wrappers that translate the Kohn-Sham eigenvalue problem to a solver-specific form, then invoke one of the solvers supported by ELSI to compute the eigenvalues, eigenvectors, or density matrix. Customizers are optional, providing a "control panel" to fine-tune all aspects of the ELSI interface layer and the solvers provided through ELSI. Utilities are subroutines that perform specialized operations not classifiable into the other families.

ELSI Controllers

elsi_init	Initializes ELSI.
elsi_set_mpi	Sets up MPI environment for ELSI.
elsi_set_blacs	Sets up BLACS environment for ELSI (when using BLACS.)
elsi_set_csc	Sets up sparsity pattern in CSC format for ELSI (when using a 1D block distributed CSC format.)
elsi_finalize	Finalizes ELSI.

Solvers Provided Through ELSI

elsi_ev_real	Solve the Kohn-Sham eigenvalue problem directly, returning eigenvectors and eigenvalues. Used for real-valued Hamiltonians stored in a dense format.
elsi_ev_complex	Solve the Kohn-Sham eigenvalue problem directly, returning eigenvectors and eigenvalues. Used for complex-valued Hamiltonians stored in a dense format.
elsi_ev_real_sparse	Solve the Kohn-Sham eigenvalue problem directly, returning eigenvectors and eigenvalues. Used for real-valued Hamiltonians stored in a sparse format. Experimental.
elsi_dm_real	Solve or circumvent the Kohn-Sham eigenvalue problem, returning the density matrix and sum of occupied eigenvalues. Output density matrix will be in a dense format.
elsi_dm_real_sparse	Solve or circumvent the Kohn-Sham eigenvalue problem, returning the density matrix and sum of occupied eigenvalues. Output density matrix will be in a sparse format. Only compatible with PEXSI at the moment.

Solver interfaces returning density matrices for complex Hamiltonians will be supported in a future release. The ELSI team is also looking into uniting elsi_dm_real and elsi_dm_real_sparse into a single subroutine (and similarly elsi_ev_real and elsi_ev_real_sparse).

ELSI Customizers

elsi_customize	Customizes the interface layer of ELSI.
elsi_customize_mu	Customizes ELSI's chemical potential determination.
elsi_customize_elpa	Customizes the ELPA solver via ELSI.
elsi_customize_omm	Customizes the libOMM solver via ELSI.
elsi_customize_pexsi	Customizes the PEXSI solver via ELSI.

ELSI Utilities

elsi_collect_pexsi	Extracts the energy density matrix and free energy density matrix from PEXSI. Only usable when PEXSI is the chosen solver and elsi_dm_real_sparse has been used.

3.1.2 Constants, Precision, and Data Types in ELSI

Constant values and choices for important runtime parameters (solver used, matrix storage format, etc.) have been defined in the `ELSI_CONSTANTS` module.

The definition for double-precision floating-point primitives is defined in the `ELSI_PRECISION` module. This and future precision definitions will utilize the `iso_c_binding` intrinsic module to maximize interoperability with C/C++ codes. We intend on moving select integer primitives (notably matrix dimensions) to 64-bit integers in a future release.

The `ELSI_CONSTANTS` and `ELSI_PRECISION` modules are public-facing, and the user is free to import them as they see fit.

Specifications for primitives in ELSI are:

real single (floating-point)	Not used
real double (floating-point)	`c_double` (Fortran), `double` primitive (C/C++)
integer	`INTEGER` primitive (Fortran), `integer` primitive (C/C++)
integer, 64-bit	Not used [planned for future release]
logical	`LOGICAL` primitive (Fortran), `integer` primitive* (C/C++)

Each component of a complex primitive has the precision of the associated real primitive, e.g. the real and imaginary components of a complex double each have the precision of a real double.

To allow multiple instances of ELSI to co-exist within a single calling code, we define an `elsi_handle` data type to encapsulate the state of the ELSI instance. For the Fortran code, this data type is defined in the `ELSI` module, and for C/C++ code this data type is defined in the `elsi.h` header file. The `elsi_handle` data type contains all runtime parameters associated with the ELSI instance, e.g. the MPI communicator, the choice of solver, etc.

An `elsi_handle` instance is initialized with the elsi_init subroutine and is subsequently passed to all other ELSI subroutine calls. Modifying the state of the ELSI instance is possible in Fortran codes by directly modifying the handle, but this is not recommended. The recommended method for modifying the state of the the ELSI instance is to use ELSI's public-facing API, as will be demonstrated in 3.1.3, 3.2, 3.3, 3.4, 3.5, and 3.6.

*For C/C++ wrapper subroutines whose Fortran equivalent passes logical arguments, we have modified the C/C++ wrapper's argument type to an `integer`, where 0 corresponds to Fortran's `.FALSE.` and any other value corresponds to `.TRUE.`.

3.1.3 Pseudo-code for Using ELSI

The typical work-flow of ELSI within a KS-DFT code is demonstrated by the following pseudo-code. For the developer desiring a "hands-on" set of examples, the source code for the ELSI regression tests (including a C/C++ example) may be found in the `/PATH/TO/ELSI/src/ELSI/test` folder.

(A)
2D block-cyclic distributed dense matrix storage (BLACS_DENSE)
ELPA solver
Eigenvalues/eigenvectors returned

Initialize SCF calculation

elsi_init(elsi_h, ELPA, parallel_mode, BLACS_DENSE, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_context, block_size)

while *SCF not converged* **do**
  Update H matrix

  elsi_customize_elpa(elsi_h, keyword=choice)
  elsi_ev_{real|complex}(elsi_h, hamiltonian, overlap, eigenvalues, eigenvectors)

  Update electron density
  Check SCF convergence
end

elsi_finalize(elsi_h)

(B)
1D block distributed compressed sparse column matrix storage (PEXSI_CSC)
ELPA solver
Eigenvalues/eigenvectors (in BLACS_DENSE format) returned

Initialize SCF calculation

elsi_init(elsi_h, solver, parallel_mode, PEXSI_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_context, block_size)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)

while *SCF not converged* **do**
    Update H matrix

    elsi_customize_elpa(elsi_h, keyword=choice)
    elsi_ev_real_sparse(elsi_h, hamiltonian, overlap, eigenvalues, eigenvectors)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)

(C)
2D block cyclic distributed dense matrix storage (BLACS_DENSE)
ELPA/libOMM/PEXSI solver
Density matrix/sum of occupied eigenvalues returned

Initialize SCF calculation

elsi_init(elsi_h, solver, parallel_mode, BLACS_DENSE, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_blacs(elsi_h, blacs_context, block_size)

while *SCF not converged* **do**
    Update H matrix

    elsi_customize_{elpa|omm|pexsi}(elsi_h, keyword=choice)
    elsi_dm_real(elsi_h, hamiltonian, overlap, density_matrix, sum_occ_eigenvalues)

    Update electron density
    Check SCF convergence
end

elsi_finalize(elsi_h)

(D)
1D block distributed compressed sparse column matrix storage (PEXSI_CSC)
PEXSI solver
Density matrix/sum of occupied eigenvalues returned

Initialize SCF calculation

elsi_init(elsi_h, PEXSI, parallel_mode, PEXSI_CSC, n_basis, n_electron, n_state)
elsi_set_mpi(elsi_h, mpi_comm)
elsi_set_csc(elsi_h, global_nnz, local_nnz, local_col, row_index, col_pointer)

while *SCF not converged* **do**

Update H matrix

elsi_customize_pexsi(elsi_h, keyword=choice)
elsi_dm_real_sparse(elsi_h, hamiltonian, overlap, density_matrix, sum_occ_eigenvalues)

Update electron density
Check SCF convergence

end

elsi_finalize(elsi_h)

3.2 ELSI Controllers

3.2.1 Initializing ELSI

The ELSI interface must be initialized via the elsi_init subroutine before any other ELSI subroutine may be called.

elsi_init(handle, solver, parallel_mode, matrix_storage_format, n_basis, n_electron, n_state)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	out	Handle to the current ELSI instance.
solver	integer	in	The desired electronic structure solver. Accepted values are: 1 (ELPA), 2 (LIBOMM), and 3 (PEXSI).
parallel_mode	integer	in	The type of parallelization used. See comment 3 for details. Accepted values are: 0 (SINGLE_PROC) and 1 (MULTI_PROC).
matrix_storage_format	integer	in	The matrix storage format used for the matrices handled by ELSI (Hamiltonian, overlap, eigenvectors, density matrix.) See comment 1 for details. Accepted values are: 0 (BLACS_DENSE) and 1 (PEXSI_CSC).
n_basis	integer	in	Number of basis functions, i.e. the global size of the Hamiltonian and overlap matrices.
n_electron	real double	in	Number of electrons.
n_state	integer	in	Number of states, as interpreted by the solver. See comment 2 for details.

Comments

1) matrix_storage_format: BLACS_DENSE(0) is a dense matrix storage format using a 2-dimensional block-cyclic distribution, i.e. the BLACS standard[14]. PEXSI_CSC(1) refers to a compressed sparse column (CSC) matrix storage format using a 1-dimensional block distribution, i.e. the PEXSI standard[15].

2) n_state: If ELPA is the chosen solver, the number of states must be equal to or larger than the number of occupied states. If libOMM is the chosen solver, the number of states must be exactly the number of occupied states, as libOMM cannot handle fractional occupation numbers[2]. PEXSI does not make use of this parameter and a dummy value may be passed.

3) parallel_mode: The two allowed values of parallel_mode, 0 (SINGLE_PROC) and 1 (MULTI_PROC), allow for three parallelization strategies commonly employed by KS-DFT codes. See comments 3a-3c.

3a) SINGLE_PROC: Solves the KS eigenproblem following a LAPACK-like fashion. This option may only be selected when a solver interface returning eigenvalues and eigenvectors is used. This allows the following parallelization strategy:

SINGLE_PROC Example:

Every MPI task independently handles a group of k-points uniquely assigned to it.

Example number of k-points: 16
Example number of MPI tasks: 4

MPI task 0 handles k-points 1, 2, 3, 4 sequentially;
MPI task 1 handles k-points 5, 6, 7, 8 sequentially;
MPI task 2 handles k-points 9, 10, 11, 12 sequentially;
MPI task 3 handles k-points 13, 14, 15, 16 sequentially.

Pseudocode 1:

```
elsi_init(..., parallel_mode=0, ...)
...
for i_kpt = 1, n_kpoints_local, 1 do
    elsi_ev_{real|complex}(elsi_h, hamiltonian_this_kpt, overlap_this_kpt, eigenvalues_this_kpt,
      eigenvectors_this_kpt)
end
```

3b+3c) MULTI_PROC: Solves the KS eigenproblem following a ScaLAPACK-like fashion. This allows the usage of the following two parallelization strategies:

MULTI_PROC Example #1:

Groups of MPI tasks coordinate to handle the same k-point, uniquely assigned to that group.

Example number of k-points: 4
Example number of MPI tasks: 16

MPI tasks 0, 1, 2, 3 cooperatively handle k-point 1;
MPI tasks 4, 5, 6, 7 cooperatively handle k-point 2;
MPI tasks 8, 9, 10, 11 cooperatively handle k-point 3;
MPI tasks 12, 13, 14, 15 cooperatively handle k-point 4.

Pseudocode 2:

```
elsi_init(..., parallel_mode=1, ...)
elsi_set_mpi(..., my_mpi_comm)
...
elsi_ev_{real|complex}(elsi_h, my_hamiltonian, my_overlap, my_eigenvalues, my_eigenvectors)
```

or

```
elsi_init(..., parallel_mode=1, ...)
elsi_set_mpi(..., my_mpi_comm)
...
elsi_dm_real(elsi_h, my_hamiltonian, my_overlap, my_density_matrix, my_sum_occ_eigenvalues)
```

MULTI_PROC Example #2:

All MPI tasks coordinate to handle each k-point, one after the other.

Example number of k-points: 4
Example number of MPI tasks: 16

MPI tasks 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 cooperatively handle k-points 1, 2, 3, 4 sequentially.

Pseudocode 3:

```
elsi_init(..., parallel_mode=1, ...)
elsi_set_mpi(..., mpi_comm_world)
...

for i_kpt = 1, n_k_points, 1 do
    elsi_ev_{real|complex}(elsi_h, hamiltonian_this_kpt, overlap_this_kpt, eigenvalues_this_kpt,
     eigenvectors_this_kpt)
end
```

3.2.2 Setting Up MPI

The MPI communicator for ELSI communication is passed into ELSI by the calling code via the elsi_set_mpi subroutine. The MPI communicator used should correspond to the parallelization strategy desired.

elsi_set_mpi(handle, mpi_comm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
mpi_comm_elsi	integer	in	MPI communicator.

3.2.3 Setting Up Matrix Storage Formats

When using the BLACS matrix storage format (BLACS_DENSE) for the matrices handled by ELSI, BLACS parameters are passed into ELSI via the elsi_set_blacs subroutine. The ELSI internal storage format requires the block sizes of the BLACS 2-dimensional block-cyclic distribution are the same in the row and column directions, as this is a requirement of ELPA. It is necessary to call this subroutine before calling elsi_ev_real, elsi_ev_complex, elsi_ev_real_sparse, or elsi_dm_real.

elsi_set_blacs(handle, blacs_context, block_size)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
blacs_context	integer	in	BLACS context.
block_size	integer	in	Block size of the 2D block-cyclic distribution, specifying both dimensions.

When using the 1D block distributed compressed sparse column matrix storage format (PEXSI_CSC) for the matrices handled by ELSI, the sparsity pattern should be passed into ELSI via the elsi_set_csc subroutine. It is necessary to call this subroutine before calling elsi_ev_real_sparse or elsi_dm_real_sparse.

elsi_set_csc(handle, global_nnz, local_nnz, local_col, row_index, col_pointer)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
global_nnz	integer	in	Global number of non-zeros.
local_nnz	integer	in	Local number of non-zeros. Specifies the dimension for the 1D block CSC format.
local_col	integer	in	Local number of matrix columns.
col_pointer	integer, rank-1 array	in	Column pointer array for the CSC storage format, in 1D block CSC format.
row_index	integer, rank-1 array	in in	Row index array for the CSC storage format. Dimensions are (local_col).

3.2.4 Finalizing ELSI

When ELSI is no longer needed (e.g. when the SCF cycle converges), the memory allocated by ELSI should be deallocated by calling elsi_finalize.

elsi_finalize(handle)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.

3.3 Solvers Provided Through ELSI

The electronic structure solvers provided through ELSI solve or circumvent the Kohn-Sham eigenvalue problem. The solver interface subroutines are differentiated based on the matrix storage format of the calling code and the desired output, *not* the implementation details of the solver (e.g. the ELPA solver may be selected when using a density-matrix-based solver interface.)

3.3.1 Solver Interfaces Returning Eigenvalues/Eigenvectors

The following subroutines return the eigenvalues and (a subset of) eigenvectors of the Hamiltonian. Only eigensolvers may be selected as the desired electronic structure solver when using these subroutines.

elsi_ev_real(handle, hamiltonian, overlap, eigenvalues, eigenvectors)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
hamiltonian	real double, rank-2 array	inout	Real Hamiltonian matrix in 2D block-cyclic dense format. See comment 1 for details.
overlap	real double, rank-2 array	inout	Real overlap matrix (or Cholesky factor) in 2D block-cyclic dense format. See comment 1 for details.
eigenvalues	real double, rank-1 array	out	Eigenvalues. Dimensions are (n_basis), as set by elsi_init.
eigenvectors	real double, rank-2 array	out	Real eigenvectors in 2D block-cyclic dense format. See comment 2 for details.

elsi_ev_complex(handle, hamiltonian, overlap, eigenvalues, eigenvectors)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
hamiltonian	complex double, rank-2 array	inout	Complex Hamiltonian matrix in 2D block-cyclic dense format. See comment 1 for details.
overlap	complex double, rank-2 array	inout	Complex overlap matrix (or Cholesky factor) in 2D block-cyclic dense format. See comment 1 for details.
eigenvalues	real double, rank-1 array	out	Eigenvalues. Dimensions are (n_basis), as set by elsi_init.
eigenvectors	complex double, rank-2 array	out	Complex eigenvectors in 2D block-cyclic dense format. See comment 2 for details.

elsi_ev_real_sparse(handle, hamiltonian, overlap, eigenvalues, eigenvectors)

NOTE: The elsi_ev_real_sparse subroutine was added late in the development cycle of the May 2017 release. While we have verified that it works for selected examples, it is considered experimental. Since the eigenvectors are not necessarily sparse, at the present moment it returns the eigenvectors in a 2D block-cycle matrix storage format (BLACS_DENSE).

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
hamiltonian	real double, rank-1 array	inout	Real Hamiltonian matrix in 1D block CSC sparse format. See comment 1 for details.
overlap	real double, rank-1 array	inout	Real overlap matrix in 1D block CSC sparse format. See comment 1 for details.
eigenvalues	real double, rank-1 array	out	Eigenvalues. Dimensions are (n_basis), as set by elsi_init.
eigenvectors	real double, rank-2 array	out	Real eigenvectors in 2D block-cyclic dense format. See comment 2 for details.

Comments

1) The Hamiltonian matrix will be destroyed by ELPA during computation. ELPA will overwrite the overlap matrix in its initial execution with the Cholesky factor, which will be reused by subsequent subroutine calls to elsi_ev_real or elsi_ev_complex. When using elsi_ev_real_sparse, the Cholesky factor (which is not sparse) is stored internally in the BLACS_DENSE format. Starting from the second call to elsi_ev_real_sparse, the input sparse overlap matrix will not be used.

2) When using the ELPA solver, elsi_ev_real, elsi_ev_complex, and elsi_ev_real_sparse compute a subset of all possible eigenvectors. The number of eigenvectors to compute is specified by the keyword n_state in elsi_init. However, the local eigenvectors array should always be initialized to correspond to a global array of size n_basis $\times$ n_basis (that is, eigenvectors should have the same BLACS descriptor as hamiltonian) to meet the internal requirements of ELPA.

3.3.2 Solver Interfaces Returning Density Matrices

The following subroutines interface to solvers to return the density matrix in the specified matrix storage format, as well as the sum of occupied single-particle energies. When the selected solver is an eigensolver, ELSI will internally construct the density matrix using the eigenvalues and eigenvectors returned by the eigensolver.

elsi_dm_real(handle, hamiltonian, overlap, density_matrix, energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
hamiltonian	real double, rank-2 array	inout	Real Hamiltonian matrix in 2D block-cyclic dense format.
overlap	real double, rank-2 array	inout	Real overlap matrix (or Cholesky factor) in 2D block-cyclic dense format. See comment 1.
density_matrix	real double, rank-2 array	out	Density matrix in 2D block-cyclic dense format.
energy	real double	out	Sum of occupied single-particle energies format.

elsi_dm_real_sparse(handle, hamiltonian, overlap, density_matrix, energy)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
hamiltonian	real double, rank-1 array	inout	Non-zero values of the real Hamiltonian matrix in 1D block CSC format.
overlap	real double, rank-1 array	inout	Non-zero values of the real overlap matrix in 1D block CSC format.
density_matrix	real double, rank-1 array	out	Non-zero values of the density matrix in 1D block CSC format.
energy	real double	out	Sum of occupied single-particle energies.

Comments

1) If the chosen solver is ELPA or libOMM, the Hamiltonian matrix will be destroyed during the computation. ELPA will overwrite the overlap matrix in its initial execution with the Cholesky factor, which will be reused by subsequent subroutine calls to elsi_dm_real.

3.4 ELSI Customizers, Abridged API

The ELSI customizers are a set of subroutines which customize the runtime parameters of ELSI. Every ELSI customizer is optional; subroutines which are required to initialize ELSI have been designated as Controllers instead. Although ELSI sets reasonable default runtime parameters for each solver, no set of default runtime parameters can adequately cover all use cases.

The API of ELSI customizers relies on optional arguments in Fortran. Customizers have the general format

elsi_customize(handle, keyword=choice)

where one or more keyword/choice pairs may be specified in a subroutine call.

In this section, we only specify the ELSI customizers and customization options of which we believe the average user should be knowledgeable, either for performance or accuracy reasons. A full API for all customizers and customization options is documented in Appendix B.

3.4.1 Customizing ELSI Interface

elsi_customize(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
no_singularity_check	logical	in	If true, the singularity check of the overlap matrix will be skipped. See comment 1 for details.	false
singularity_tolerance	real double	in	Eigenfunctions of the overlap matrix with eigenvalues smaller than this threshold will be removed to avoid ill-conditioning. See comment 1 for details.	10^{-5}
stop_singularity	logical	in	If true, the code always stops if the overlap matrix is singular. See comment 1 for details.	false

1) If the singularity check is not disabled, in the first iteration of each SCF cycle, possible singularity of the overlap matrix is checked by computing all its eigenvalues. If there is any eigenvalue smaller than singularity_tolerance, the matrix is considered singular. If stop_singularity is true, the code will stop upon determining that the overlap matrix is singular. If stop_singularity is false, the ELPA and libOMM solvers will continue with an alternative algorithm (i.e. Cholesky factorization is not used), transforming the generalized eigenproblem to the standard form. The PEXSI solver will use the full, singular overlap matrix.

3.4.2 Customizing libOMM solver

elsi_customize_omm(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
n_elpa_steps	integer	in	Number of ELPA steps before libOMM. See comment 1 for details.	3
omm_flavor	integer	in	Method to perform OMM minimization. See comment 2 for details.	2
omm_tolerance	real double	in	Tolerance of orbital minimization.	10^{-10}

Comments

1) n_elpa_steps: libOMM and ELSI developers have found that libOMM is optimal at later stages of the SCF cycle where the electronic structure is closer to its expected local minimum, requiring only one CG iteration per libOMM call. Accordingly, we have chosen to use ELPA initially when libOMM is selected as the solver, only switching to libOMM once a set number of SCF cycles (specified by n_elpa_steps) has been computed. This is an enhancement over the original usage of a random guess for the starting electronic structure used by libOMM, which leads to a large number of CG iterations to converge OMM in early SCF steps.

2) omm_flavor: Allowed choices are 0 for basic minimization of a generalized eigenproblem and 2 for a Cholesky factorization of the overlap matrix transforming the generalized eigenproblem to the standard form.

3.4.3 Customizing PEXSI solver

elsi_customize_pexsi(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
n_poles	integer	in	Number of poles used by PEXSI. See comment 1.	40
n_procs_per_pole	integer	in	Number of MPI tasks assigned to each pole. See comment 2.	n_procs/n_poles
n_electron_accuracy	real double	in	Tolerance for error in electron count, used as stopping criterion for PEXSI.	0.01

Comments

1) n_poles: Standard KS-DFT calculations typically require 40∼80 poles. Perform a convergence test if in doubt.

2) n_procs_per_pole: To enable the efficient pole parallelization of PEXSI, the ELSI dense density matrix solver only accepts n_procs_per_pole = n_procs/n_poles. Any other input value will be ignored. The input and output matrices of the dense density matrix solver are distributed among all available processors in a 2D block-cyclic manner. More flexibility is offered with the ELSI sparse density matrix solver, which only requires n_procs_per_pole = n_procs/p, where p is a positive integer. The underlying parallelization, depending on p, is summarized in Table 3.1. The input and output matrices of the sparse density matrix solver are always distributed among the first n_procs_per_pole processors in a 1D block distribution, independent of the actual value of p.

Table 3.1: Pole parallelization utilized in PEXSI depending on the number of MPI tasks and the number of poles.
n_procs: total number of MPI tasks.
n_poles: number of poles.
n_procs_per_pole: number of MPI tasks per pole.
p: n_procs/n_procs_per_pole.
q: the smallest integer greater than n_poles/p.

Condition	Pole parallelization	Description
p = n_poles	full	all poles computed in parallel
1 < p < n_poles	partial	q groups of poles computed in parallel
p = 1	none	all poles computed in serial

3.5 ELSI Utilities

ELSI utility subroutines provide miscellaneous features that do not fit cleanly into any of the other categories, but may provide necessary features for KS-DFT codes.

3.5.1 Collect Additional Information from PEXSI

The elsi_collect_pexsi subroutine collects results not accessible via the ELSI solver interface from a PEXSI calculation: the chemical potential, the energy density matrix, and the free energy density matrix. It must be called after PEXSI has been used via elsi_dm_real_sparse.

elsi_collect_pexsi(handle, chemical_potential, energy_dm, free_energy_dm)

Argument	Data Type	in/out	Explanation
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.
chemical_potential	real double	out	Chemical potential/Fermi level.
energy_dm	real double, rank-1 array	out	Non-zero values of the energy density matrix in 1D block CSC format.
free_energy_dm	real double, rank-1 array	out	Non-zero values of the free energy density matrix in 1D block CSC format.

3.6 C/C++ Interface

ELSI is written in Fortran, but many KS-DFT codes use C/C++. ELSI provides a C/C++ wrapper around the core Fortran ELSI code. In this section, we list the C/C++ wrapper functions. Each C/C++ wrapper function corresponds to a Fortran subroutine, where we have consistently prefixed the original Fortran name with c_ to avoid namespace collisions. Argument lists are identical to the associated native Fortran subroutine. Information on conversion between Fortran and C/C++ primitive types may be found in Section 3.1.2. We recommend that the user looks at the `elsi.h` header file directly for information when to pass a variable by reference or by value.

ELSI Controllers (C version)

c_elsi_init	Initializes ELSI Interface.
c_elsi_set_mpi	Sets up MPI environment for ELSI Interface. See comment 1.
c_elsi_set_blacs	Sets up BLACS environment for ELSI Interface.
c_elsi_set_csc	Sets up sparsity pattern for ELSI Interface in 1D block distributed CSC format.
c_elsi_finalize	Finalizes ELSI Interface.

Comments

1) Since ELSI is written in Fortran, it may be necessary to call MPI_Comm_c2f before c_elsi_set_mpi to make the C/C++ handle of MPI communicator recognizable in Fortran code.

Solvers Provided Through ELSI (C version)

c_elsi_ev_real	Invokes ELSI real eigensolver.
c_elsi_ev_complex	Invokes ELSI complex eigensolver.
c_elsi_ev_real_sparse	Invokes ELSI real sparse eigensolver.
c_elsi_dm_real	Invokes ELSI real density matrix solver.
c_elsi_dm_real_sparse	Invokes ELSI real sparse density matrix solver.

ELSI Customizers (C version)

Note: Owing to the lack of optional variables in C, *all* arguments must be specified for C/C++ wrapper functions for ELSI Customizers.

c_elsi_customize	Customizes ELSI Interface.
c_elsi_customize_mu	Customizes ELSI chemical potential determination.
c_elsi_customize_elpa	Customizes ELPA solver.
c_elsi_customize_omm	Customizes libOMM solver.
c_elsi_customize_pexsi	Customizes PEXSI solver.

ELSI Utilities (C version)

c_elsi_collect_pexsi	Extracts the energy density matrix and free energy density matrix from PEXSI. Only usable when c_elsi_dm_real_sparse has been used with PEXSI.

3.7 Python Interface [Future]

A Python interface is planned for a future release. We will update this section when appropriate.

A Full List of Keywords in the make.sys File

A.1 General Flags

MPIFC	= MPI Fortran compiler [default: empty]
MPICC	= MPI C compiler [default: empty]
MPICXX	= MPI C++ compiler [default: empty]
LINKER	= Linker [default: $(MPIFC)]
FFLAGS	= Fortran compiler flags [default: empty]
CFLAGS	= C compiler flags [default: empty]
CXXFLAGS	= C++ compiler flags [default: empty]
LDFLAGS	= Linker flags [default: $(FFLAGS_I)]
ARCHIVEFLAGS	= Archiver (ar) flags [default: cr]
CXX_LIB	= Standard C++ library, e.g. -lstdc++ [default: empty]
SCALAPACK_LIB	= ScaLAPACK libraries [default: empty]

A.2 ELSI Interface Flags

MPI_EXEC	= Name of MPI executable [default: mpirun]
TOMATO_SEED	= Path to TOMATO-SEED folder [default: empty]
FFLAGS_I	= Fortran compiler flags for ELSI Interface [default: $(FFLAGS)]
C_INTERFACE	= Create interface to C? [default: yes]

A.3 ELPA Flags

ELPA2_KERNEL	= Choice of ELPA2 kernels [default: Generic]
FFLAGS_E	= Fortran compiler flags for ELPA [default: $(FFLAGS)]
CFLAGS_E	= C compiler flags for ELPA [default: $(CFLAGS)]
EXTERNAL_ELPA	= Use external, precompiled ELPA? [default: no]
ELPA_LIB	= ELPA library flag(s) [default: built-in version]
ELPA_INC	= ELPA include flag(s) [default: built-in version]

A.4 libOMM Flags

FFLAGS_O	= Fortran compiler flags for libOMM [default: $(FFLAGS)]
EXTERNAL_OMM	= Use external, precompiled libOMM? [default: no]
OMM_LIB	= libOMM library flag(s) [default: built-in version]
OMM_INC	= libOMM include flag(s) [default: built-in version]

A.5 PEXSI Flags

CFLAGS_P	= C compiler flags for PEXSI [default: $(CFLAGS)]
CXXFLAGS_P	= C++ compiler flags for PEXSI [default: $(CXXFLAGS)]
METIS_LIB	= METIS library flags [default: empty]
PARMETIS_LIB	= ParMETIS library flags [default: empty]
SUPERLU_LIB	= SuperLU_Dist library flags [default: empty]
SUPERLU_INC	= SuperLU_Dist include flags [default: empty]
EXTERNAL_PEXSI	= Use external, precompiled PEXSI? [default: no]
PEXSI_LIB	= PEXSI library flag(s) [default: built-in version]
PEXSI_INC	= PEXSI include flag(s) [default: built-in version]

B ELSI Customizers, Complete API

In this Appendix, we lay out the full API for the ELSI Customizers. An abridged version giving the essential information may be found in Section 3.4.

B.1 Customizing ELSI Interface

elsi_customize(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
print_detail	logical	in	If true, print detailed information.	false
overlap_is_unit	logical	in	If true, the overlap matrix is assumed to be the identity matrix. See comment 1 for details.	false false
numerical_zero	real double	in	Threshold to define numeric "zero".	10^{-15}
no_singularity_check	logical	in	If true, the singularity check of overlap matrix will be skipped. See comment 2 for details.	false
singularity_tolerance	real double	in	Eigenfunctions of the overlap matrix with eigenvalues smaller than this threshold will be removed to avoid ill-conditioning. See comment 2 for details.	10^{-5}
stop_singularity	logical	in	If true, the code always stops if the overlap matrix is singular. See comment 2 for details.	false
uplo	integer	in	Reserved for future functionality. Must be set to 0 in ELSI May 2017 release. See comment 3 for details.	0

Comments

1) overlap_is_unit: no_singularity_check, singularity_tolerance, and stop_singularity will be ignored when overlap_is_unit is true.

2) If the singularity check is not disabled, in the first iteration of each SCF cycle, possible singularity of the overlap matrix is checked by computing all its eigenvalues. If there is any eigenvalue smaller than singularity_tolerance, the matrix is considered singular. If stop_singularity is true, the code will stop upon determining that the overlap matrix is singular. If stop_singularity is false, the ELPA and libOMM solvers will continue with an alternative algorithm (i.e. Cholesky factorization is not used), transforming the generalized eigenproblem to the standard form. The PEXSI solver will use the full, singular overlap matrix.

3) uplo: The only allowed value currently is FULL_MAT (0). UT_MAT (1) and LT_MAT (2), representing upper triangular and lower triangular respectively, will be supported in the next release of ELSI.

B.2 Customizing ELSI Chemical Potential Determination

elsi_customize_mu(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
broadening_scheme	integer	in	Broadening scheme in chemical potential determination. Accepted values are: GAUSSIAN(0), FERMI(1), METHFESSEL-PAXTON 0th order(2), 1st order(3).	0
broadening_width	real double	in	Broadening width used in chemical potential determination.	10^{-2}
occ_accuracy	real double	in	Tolerance of electron count error in chemical potential and occupation number determination.	10^{-10}
mu_max_steps	integer	in	Maximum bisection steps to determine the chemical potential.	100
spin_degeneracy	real double	in	Only 2.0 is supported (non-spin-polarization).	2.0

B.3 Customizing ELPA Solver

elsi_customize_elpa(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
elpa_solver	integer	in	Use ELPA 1-stage (1) or 2-stage solver (2).	1 if n_basis $<$ 250 2 otherwise

B.4 Customizing libOMM Solver

elsi_customize_omm(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
n_elpa_steps	integer	in	Number of ELPA steps before libOMM. See comment 1 for details.	3
omm_flavor	integer	in	Method to perform OMM minimization. See comment 2 for details.	2
eigen_shift	real double	in	Eigenspectrum shift parameter. See comment 3 for details.	0.0
omm_tolerance	real double	in	Tolerance of orbital minimization.	10^{-10}
use_pspblas	logical	in	If true, use pspBLAS sparse linear algebra. See comment 4 for details.	false
omm_output	logical	in	If true, output details in OMM calculation.	false

Comments

1) n_elpa_steps: libOMM and ELSI developers have found that libOMM is optimal at later stages of the SCF cycle where the electronic structure is closer to its expected local minimum, requiring only one CG iteration per libOMM call. Accordingly, we have chosen to use ELPA initially when libOMM is selected as the solver, only switching to libOMM once a set number of SCF cycles (specified by n_elpa_steps) has been computed. This is an enhancement over the original

usage of a random guess for the starting electronic structure used by libOMM, which leads to a large number of CG iterations to converge OMM in early SCF steps.

2) omm_flavor: Allowed choices are 0 for basic minimization of a generalized eigenproblem and 2 for a Cholesky factorization of the overlap matrix transforming the generalized eigenproblem to the standard form.

3) eigen_shift: Can be set to 0 in most cases.

4) use_pspblas: pspBLAS (Parallel SParse BLAS) library performs sparse matrix multiplications in parallel. This is an EXPERIMENTAL feature for test purpose only.

B.5 Customizing PEXSI Solver

elsi_customize_pexsi(handle, keyword=choice)

Argument	Data Type	in/out	Explanation	Default
handle	type(elsi_handle)	inout	Handle to the current ELSI instance.	
temperature	real double	in	Energy E that correlates to electron temperature via $T = E/k_B$, in the same units as the Hamiltonian.	0.0
gap	real double	in	Spectral gap, can be set to 0 in most cases. See comment 1 for details.	0.0
delta_e	real double	in	Upper bound for the spectral radius of $S^{-1}H$.	10
n_poles	integer	in	Number of poles used by PEXSI.	40
n_procs_per_pole	integer	in	Number of MPI tasks assigned to one PEXSI pole. See comment 3 for details.	n_procs/n_poles
max_iteration	integer	in	Maximum number of PEXSI iterations after each inertia counting procedure.	3
mu_min	real double	in	Initial guess for lower bound of chemical potential.	-10.0
mu_max	real double	in	Initial guess for upper bound of chemical potential.	10.0
mu0	real double	in	Initial guess for the chemical potential. See comment 4 for details.	0.0
mu_inertia_tolerance	real double	in	Stopping criterion, in terms of the chemical potential, for the inertia counting procedure.	0.05
mu_inertia_expansion	real double	in	If the chemical potential is not in the initial interval, the interval is expanded by this value.	0.3
mu_safeguard	real double	in	Safeguard criterion for the chemical potential to reinvoke the inertia counting procedure.	0.05
n_electron_accuracy	real double	in	Tolerance for error in electron count, used as stopping criterion for PEXSI.	0.01
matrix_type	integer	in	Only 0 is supported (real, symmetric).	0
is_symbolic_factorize	integer	in	Whether to perform the symbolic factorization. 0- No; 1- Yes.	1 in the first SCF step; 0 afterwards
ordering	integer	in	Only 0 is supported (Parallel ordering using ParMETIS).	0
np_symbolic_factorize	integer	in	Number of processors to be used for symbolic factorization. See comment 5 for details.	1
verbosity	integer	in	Level of output information. 0- No; 1- Basic; 2- Detailed.	1

Comments

1) gap: This parameter is only used to accelerate the convergence of PEXSI. The existence of a spectral gap is not required by PEXSI solver. In principle PEXSI works for insulators, semiconductors, and metals [5, 6].

2) n_poles: KS-DFT calculations typically require 40∼80 poles. Perform a convergence test if in doubt.

3) n_procs_per_pole: To enable the efficient pole parallelization of PEXSI, ELSI dense density matrix solver only accepts n_procs_per_pole = n_procs/n_poles. Any other input value will be ignored. The input and output matrices of the dense

density matrix solver are distributed among all available processors in a 2D block-cyclic manner. More flexibility is offered with ELSI sparse density matrix solver, which only requires n_procs_per_pole = n_procs/p, where p is a positive integer. The underlying parallelization, depending on p, is summarized in Table 3.1. The input and output matrices of the sparse density matrix solver are always distributed among the first n_procs_per_pole processors in a 1D block manner, independent of the actual value of p.

4) mu0: Starting from the second SCF iteration, the chemical potential from the previous step will be used as the initial guess of the current step, regardless of the choice of mu0.

5) np_symbolic_factorize: The number of processors for symbolic factorization cannot be larger than a magic number depending on the matrix and the machine. Therefore, the default value of np_symbolic_factorize is set to 1. It is worth testing and increasing this number.

Bibliography

[1] W. Kohn and L.J. Sham, Self-consistent equations including exchange and correlation effects, Physical Review, 140, 1133-1138 (1965).

[2] F. Corsetti, The orbital minimization method for electronic structure calculations with finite-range atomic basis sets, Computer Physics Communications, 185, 873-883 (2014).

[3] T. Auckenthaler et al., Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations, Parallel Computing, 37, 783-794 (2011).

[4] A. Marek et al., The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science, Journal of Physics: Condensed Matter, 26, 213201 (2014).

[5] L. Lin et al., Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems, Communications in Mathematical Sciences, 7, 755-777 (2009).

[6] L. Lin et al., Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion, Journal of Physics: Condensed Matter, 25, 295501 (2013).

[7] S. Mohr et al., Efficient computation of sparse matrix functions for large scale electronic structure calculations: The CheSS library, arXiv:1704.00512 (2017).

[8] M. Keceli et al., Shift-and-invert parallel spectral transformation eigensolver: massively parallel performance for density-functional based tight-binding, Journal of Computational Chemistry, 37, 448-459 (2016).

[9] W. Hu et al., DGDFT: A massively parallel method for large scale density functional theory calculations, The Journal of Chemical Physics, 143, 124110 (2015).

[10] V. Blum et al., Ab initio molecular simulations with numeric atom-centered orbitals, Computer Physics Communications, 180, 2175-2196 (2009).

[11] M. Valiev et al., NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations, Computer Physics Communications, 181, 1477-1489 (2010).

[12] J.M. Soler et al., The SIESTA method for ab initio order-N materials simulation, Journal of Physics: Condensed Matter, 14, 2745-2779 (2002).

[13] S. Mohr et al., Accurate and efficient linear scaling DFT calculations with universal applicability, Physical Chemistry Chemical Physics, 17, 31360-31370 (2015).

[14] http://www.netlib.org/blacs

[15] http://math.berkeley.edu/∼linlin/pexsi/page_data_type.html

License and Copyright

ELSI Interface software is licensed under the 3-clause BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3) Neither the name of the "ELectronic Structure Infrastructure (ELSI)" project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The source code of ELPA (LGPL), libOMM (2-clause BSD), and PEXSI (3-clause BSD) are redistributed within this ELSI release. Individual license of each library can be found in the corresponding subfolder.