ELPA Manual

User's Guide and Best Practices

Version 2023.05.001

A. Marek, T. Melson, S. Soltani, P. Karpov
The ELPA developer team
Max Planck Computing and Data Facility

September 22, 2023

Contents

1 About ELPA

The computation of a subset or all eigenvalues and eigenvectors of a Hermitian matrix has high relevance for various scientific disciplines. Typically, direct solvers are used for the calculation of a significant part of the eigensystem. For large problems, solving for the eigensystem with the existing solvers can become the computational bottleneck.

With the aim of developing and implementing an efficient eigenvalue solver for petaflop applications, ELPA (Eigenvalue soLvers for Petaflop Applications) was born, and today it has become a modern library for direct, efficient, and scalable solution of eigenvalue problems involving dense, Hermitian matrices.

The ELPA library was originally created by the ELPA consortium consisting of the following organizations:

- Max Planck Computing and Data Facility (MPCDF), formerly known as Rechenzentrum Garching der Max-Planck-Gesellschaft (RZG),
- Bergische Universität Wuppertal, Lehrstuhl für angewandte Informatik,
- Technische Universität München, Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen,
- Fritz-Haber-Institut der Max-Planck-Gesellschaft, Berlin, Abt. Theorie,
- Max-Plack-Institut für Mathematik in den Naturwissenschaften, Leipzig, Abt. Komplexe Strukturen in Biologie und Kognition, and
- IBM Deutschland GmbH

ELPA uses the distributed matrix layout of ScaLAPACK, but replaces the solution steps with subroutines of its own. Two variants of the solver are available: a one-step, and a two-step solver hereinafter referred to as ELPA1 and ELPA2, respectively.

1.1 How to obtain ELPA

ELPA is an open source project. Its source code is freely available at `https://gitlab.mpcdf.mpg.de/elpa/elpa`. It is distributed under the terms of the GNU Lesser General Public License version 3 as published by the Free Software Foundation. A mirror of the above repository is also available on GitHub, which is mainly for opening issues and merge requests as well as contributions from the developer's community: `https://github.com/marekandreas/elpa`. Additionally, ELPA can be obtained from the following sources:

- Official release tarball from the ELPA webpage
- As a packaged software for several Linux distributions (e.g., Debian, Fedora, OpenSuse)

1.2 Terms of use

ELPA can be freely obtained, used, modified and redistributed under the terms of the GNU Lesser General Public License version 3.

No other conditions have to be met. Nonetheless, we would be grateful if you consider citing the following articles:

1. If you use ELPA in general:

- T. Auckenthaler, V. Blum, H. J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations", Parallel Computing 37, 783–794 (2011). doi:10.1016/j.parco.2011.05.002.
- A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H. J. Bungartz, and H. Lederer, "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science", Journal of Physics Condensed Matter, 26 (2014) doi:10.1088/0953-8984/26/21/213201

2. If you use the GPU version of ELPA:
   - P. Kus, A. Marek, and H. Lederer, "GPU Optimization of Large-Scale Eigenvalue Solver", In: F. Radu, K. Kumar, I. Berre, J. Nordbotten, I. Pop (eds) Numerical Mathematics and Advanced Applications ENUMATH 2017. ENUMATH 2017. Lecture Notes in Computational Science and Engineering, vol 126. Springer, Cham
   - V. Yu, J. Moussa, P. Kus, A. Marek, P. Messmer, M. Yoon, H. Lederer, and V. Blum, "GPU-Acceleration of the ELPA2 Distributed Eigensolver for Dense Symmetric and Hermitian Eigenproblems", Computer Physics Communications, 262, 2021
3. If you use the new API and/or autotuning:
   - P. Kus, A. Marek, S. S. Koecher, H. H. Kowalski, Ch. Carbogno, Ch. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the Eigenvaluesolvers in the ELPA Library", Parallel Computing 85, 167–177 (2019)
4. If you use the new support for skew-symmetric matrices:
   - P. Benner, C. Draxl, A. Marek, C. Penke, and C. Vorwerk, "High Performance Solution of Skew-symmetric Eigenvalue Problems with Applications in Solving the Bethe-Salpeter Eigenvalue Problem", `https://arxiv.org/abs/1912.04062`, submitted to Parallel Computing

1.3 Current release

The current ELPA release is 2023.05.001. It supports the API version 20231705. The oldest API version supported by the current release is version 20170403. On more information on the API versions, please have a look at Subsection 5.1

2 Quick start guide

This section gives a very short overview for the impatient users on how to use ELPA from a Fortran, C, or C++ application. Before showing the respective examples, a few things should be noted:

1. It is assumed that the ELPA library is already installed on your system, either by a system administrator or by you via a system package or manually. For manual installation, please have a look at the Section 3.
2. It is assumed that you can link your application against the installed ELPA library. If you need instructions on how to do that, please have a look at the Section 4.
3. The examples provide a simple explanation on how to use ELPA within your application. They neglect all options about tailoring ELPA to your specific needs and about how to achieve the best performance possible. For quite a number of control options, we have chosen *reasonable* but maybe *not perfect* defaults. If you want to tune the usage and the performance of ELPA to your specific needs, please have a look at the Section 5.2 on key-value pairs and at the Section 6.1 on autotuning.

To use the ELPA library in your code, follow these steps:

1. Include the header files (for C or C++ applications) or use the ELPA module (for Fortran applications).
2. Create the matrix for which the (generalized) eigenvalue problem should be solved. Note that in case of an MPI application, all matrices must be distributed in a BLACS block-cyclic distribution, as it must be done for ScaLAPACK.
3. Define a handle for an ELPA object.
4. Initialize the ELPA library.
5. Allocate the ELPA object.
6. Set the *mandatory parameters*, namely as the matrix size, the number of eigenvectors to be calculated, the block size of the BLACS block-cyclic distribution of the matrix, and additional parameters for the MPI setup. Note that these parameters are *fixed* for the lifetime of the ELPA object.
7. Set up the ELPA object via the `setup` method.
8. Set some *runtime options*.
   - If GPUs should be used: Set one of the keywords `nvidia-gpu`, `amd-gpu`, or `intel-gpu` via the ELPA `set` method and call the `set_gpu_parameters` method.
   - Set any other combination of *runtime options* (see Sec. 5.2) to control the ELPA runtime behavior to your preference.
9. Call one of ELPA's math functions. Examples for most commonly used routines are `eigenvectors`, `eigenvalues`, `generalized_eigenvectors`, and `generalized_eigenvalues` (see Sec. 5.3).
10. Change the *runtime options* and call the same or other ELPA math functions as often as you wish, as long as the *mandatory parameters* from above are kept constant and still apply to your problem.
11. When finished: Deallocate the ELPA object and unintialize the ELPA library.

Below we provide *minimalistic* examples of how to call ELPA from a Fortran or C/C++ application. These examples are, however, not self-contained. They will only compile and run if they are embedded into an existing MPI application. Also the BLACS setup and matrix allocation must happen before. For standalone examples, see the test programs in the `test` directory of ELPA's source code.

Please note that in the examples below, we first set the *mandatory parameters* (Sec. 5.2.1), initialize ELPA object, and then set the *runtime options* (Sec. 5.2.2). For the runtime options we set only GPU-related ones, but no other options for control and tuning of the ELPA library. See the discussion at the beginning of Sec. 5.2) for the difference between the mandatory parameters and the runtime options.

Also note that the *ELPA API version* is set to 20231705. The API version defines the set of key-value pairs which can be used to control the ELPA library and also defines the procedures provided by the library. For more details please have a look at Section 5.1.

2.1 Fortran example

```fortran
use elpa

class(elpa_t), pointer :: elpaInstance
integer :: success
! We urge the user to always check the error codes of all ELPA functions

if (elpa_init(20231705) /= ELPA_OK) then
    print *, "ELPA API version not supported"
    ! Handle this error in your application
endif

elpaInstance => elpa_allocate(success)
! Check success code, e.g. with
if (success /= ELPA_OK) then
    print *, "Could not allocate ELPA instance"
    ! Handle this error in your application
endif

! Set mandatory parameters describing the matrix and its MPI distribution
call elpaInstance%set("na", na, success)
if (success /= ELPA_OK) then
    print *, "Could not set parameter na"
    ! Handle this error in your application
endif
call elpaInstance%set("nev", nev, success)
! Check success code ...
call elpaInstance%set("local_nrows", na_rows, success)
! Check success code ...
call elpaInstance%set("local_ncols", na_cols, success)
! Check success code ...
call elpaInstance%set("nblk", nblk, success)
! Check success code ...
call elpaInstance%set("mpi_comm_parent", MPI_COMM_WORLD, success)
! Check success code ...
```

```fortran
call elpaInstance%set("process_row", my_prow, success)
! Check success code ...
call elpaInstance%set("process_col", my_pcol, success)
! Check success code ...

! Set up the elpa object
success = elpaInstance%setup()
if (success /= ELPA_OK) then
    print *, "Could not setup the ELPA object"
    ! Handle this error in your application
endif

! Set runtime options, e.g. GPU settings
call elpaInstance%set("nvidia-gpu", 1, success) ! 1=on, 0=off
! Check success code ...

! In case of GPU usage you have the choice whether ELPA
! should automatically assign each MPI task to a certain GPU
! (this is default) or whether you want to set this assignment
! for *each* task yourself.
! This is how you set the assignment yourself,
! for example in a round-robin fashion
my_gpu_id = mod(myrank, number_of_GPU_devices_per_node)
call elpaInstance%set("use_gpu_id", my_gpu_id, success)
! Check success code ...

! Finalize the GPU setup
call set_gpu_parameters()

! If desired, set other tunable runtime options...

! Solve the eigenvalue problem to obtain eigenvalues and eigenvectors
call elpaInstance%eigenvectors(a, ev, z, success)
! Check success code ...

! Cleanup
call elpa_deallocate(elpaInstance, success)
! Check success code ...

call elpa_uninit()
```

2.2 C/C++ example

```c
#include <elpa/elpa.h>

elpa_t elpaInstance;
int success;
// We urge the user to always check the error code of all ELPA functions

if (elpa_init(20231705) != ELPA_OK) {
    fprintf(stderr, "ELPA API version not supported");
    // Handle this error in your application
```

```
}

elpaInstance = elpa_allocate(&success);
if (success != ELPA_OK) {
    fprintf(stderr, "Could not allocate ELPA instance");
    // Handle this error in your application
}

// Set mandatory parameters describing the matrix and its MPI distribution
elpa_set(elpaInstance, "na", na, &success);
// Check success code ...
elpa_set(elpaInstance, "nev", nev, &success);
// Check success code ...
elpa_set(elpaInstance, "local_nrows", na_rows, &success);
// Check success code ...
elpa_set(elpaInstance, "local_ncols", na_cols, &success);
// Check success code ...
elpa_set(elpaInstance, "nblk", nblk, &success);
// Check success code ...
elpa_set(elpaInstance, "mpi_comm_parent", MPI_COMM_WORLD, &success);
// Check success code ...
elpa_set(elpaInstance, "process_row", my_prow, &success);
// Check success code ...
elpa_set(elpaInstance, "process_col", my_pcol, &success);
// Check success code ...

// Set up the elpa object
success = elpa_setup(elpaInstance);
if (success != ELPA_OK) {
    fprintf(stderr, "Could not set up the ELPA object");
    // Handle this error in your application
}

// Set runtime options, e.g. GPU settings
elpa_set(elpaInstance, "nvida-gpu", 1, &success); // 1=on, 0=off
// Check success code ...

// In case of GPU usage you have the choice whether ELPA
// should automatically assign each MPI task to a certain GPU
// (this is default) or whether you want to set this assignment
// for *each* task yourself.
// This is how you set the assignment yourself,
// for example in a round-robin fashion
my_gpu_id = myrank%number_of_GPU_devices_per_node;
elpa_set(elpaInstance, "use_gpu_id", my_gpu_id, &success);
// Check success code ...

// Finalize the GPU setup
set_gpu_parameters();

// If desired, set other tunable runtime options...
```

```
// Solve the eigenvalue problem to obtain eigenvalues and eigenvectors
elpa_eigenvectors(elpaInstance, a, ev, z, &success);
// Check success code ...

// Cleanup
elpa_deallocate(elpaInstance, &success);
// Check success code ...

elpa_uninit();
```

3 Installation guide

The build system of ELPA is the standard GNU autotools (autoconf and automake installation infrastructure) and consists of the following steps:

- `./configure`
- `make`
- `make check`
- `make install`

Note that the `configure` script is included in the official ELPA release tarballs, which can be obtained from the ELPA website. The `configure` script most likely **not** included if you obtain the ELPA sources by other means, in particular, if you use a git clone of the ELPA repository or you obtain a tarball from the git repository. To generate the `configure` in such cases, you must run the shell script `./autogen.sh`.

We describe the ELPA dependencies in Section 3.1 and then elaborate on the individual installation steps in Sections 3.2-3.3. A minimal complete installation example is given in Section 3.4. Finally, Section 3.5 gives some hints on the installation troubleshooting.

3.1 Dependencies

In order to build ELPA the following prerequisites and dependencies must be met:

1. Build tools:
   The *autoconf*, *automake* and *libtool* build tools must be installed
2. Compilers:
   ELPA is written in Fortran, C and C++. The GPU versions are written in CUDA, HIP, or SYCL. Thus you need several compilers to build ELPA.
   (a) Fortran compiler: a recent Fortran compiler is needed. It must fully support the Fortran 2003 and parts of the Fortran 2008 standard. To achieve the best performance if possible the most recent compilers should be used.
   (b) C compiler: a recent C compiler is needed. The compiler must at least support the C11 standard
   (c) C++ compiler: a recent C compiler is needed. The compiler must at least support the C++11. **Note** that for build with SYCL, the C++17 standard must be supported and used.

In case of GPU build, additional compilers are needed, depending on which version of the GPU support should be build. We recommend using the most recent as possible GPU software stack version.

- Nvidia-GPUs: The CUDA software stack must be installed and the `nvcc` compiler is needed.
- AMD-GPUs: The ROCm software stack must be installed and the `hip` compiler is needed.
- Intel-GPUs: The Intel oneAPI compilers `icx` and `ifx` must be installed.

3. External libraries: Some external libraries are needed at build and runtime:

   - the Basic Linear Algebra Subroutines (BLAS)
   - the Linear Algebra Package (LAPACK)

   If ELPA is build for parallel distributed runs (which is the preferred case), in addition

   - the Basic Linear Algebra Communication Subroutines (BLACS)
   - ScaLAPACK
   - Message Passing Interface (MPI)

   are needed. Depending on whether you want to build ELPA in the Nvidia, AMD, or Intel GPU version, some additional libraries might be needed:

   - Nvidia-GPUs: cublas, cusolver and potentially NCCL
   - AMD-GPU: rocblas and rocsolver
   - Intel-GPUs: oneAPI MKL

ELPA can be configured to run sequentially as well as in parallel on shared- and/or distributed-memory systems. The shared-memory parallel algorithm uses OpenMP threads, while the programming model of the distributed algorithm is based on the message passing library (MPI). In addition, the hybrid models MPI+OpenMP, MPI+GPU, and MPI+OpenMP+GPU are also supported. For details on the installation process and the necessary configure options, please see Sec. 3.2. **Note** that the `sequential` build option is only meant for installation on desktop or laptop machines. Such a build of ELPA provides you the full API, such that you can develop applications with ELPA, but obviously the performance of such a build will be very suboptimal.

3.2 Configuration

Running the `configure` script is the first step of the installation procedure. Note that if this script is not present in the ELPA root folder, e.g. if you obtained the ELPA sources from a git clone as in the example below, it can be easily created using `autogen.sh`, otherwise you can skip the following step.

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa
./autogen.sh
```

It is best practice to run the configuration in a subdirectory in order to keep the source directory clean:

```
mkdir build
cd build
../configure [options]
```

3.2.1 Compiler and linker variables for configure

We observe that most problems with building ELPA arise from a misunderstanding how to pass flags to the compilers, the linker and how to specify the link line for the libraries which ELPA needs as external dependencies. Thus we want to mention here how one can typically control these when calling a configure script.

Note these variables represent a very generic concept which applies to all builds with `autoconf` tools, independent of the ELPA library.

`FC`	The Fortran compiler to use. **Examples:** `FC="mpif90"` (for gcc) or `FC="mpiifort"` (for Intel) Note, that this variable must point to the Fortran compiler executable you want to envoke. Thus in case of an MPI build it must be the MPI Fortran compiler, in case of a serial build – the Fortran compiler.
`CC`	The C compiler to use. textbfExamples: `CC="mpigcc"` or `CC="mpiicc"`
`CXX`	The C++ compiler to use. **Examples:** `CXX="mpicxx"` or `CXX="mpiicpc"`
`FCFLAGS`	All flags that must be passed to the Fortran compiler to control the compiler's behavior. **Example:** `FCFLAGS="-O2 -mavx"` Note, that setting the Fortran optimization and vectorization flags via FCFLAGS (as well C/C++ flags via CFLAGS/CXXFLAGS, see below) is of **utmost** importance to obtain a good ELPA performance. We elaborate on the optimization and vectorization flags in Sec. 3.2.2.
`CFLAGS`	All flags that must be passed to the C compiler to control the compiler's behaviour. **Example:** `CFLAGS="-O2 -mavx"`
`CXXFLAGS`	All flags which must be passed to the C++ compiler to control the compiler's behaviour. **Example:** `CXXFLAGS="-O2 -mavx"`
`LDFLAGS`	All flags which must be passed to the linker to control the linkers's behaviour. **Example:** `LDFLAGS=" -Wl,-rpath,/absolute_path_to_a_library"`
`LIBS`	External libraries that you want to link against. **Example:** `LIBS=" -Wl,-L/absolute_path_to_a_library -llibrary"`

In addition to these standard variables of `autotools`, the ELPA configure honors some special variables:

`SCALAPACK_FCFLAGS`	Additional Fortran compiler flags for ScaLAPACK usage. **Example:** `SCALAPACK_FCFLAGS="-I$MKL_HOME/include/intel64/lp64"`

Note, that this variable is a convenience feature.
You can also pass these flags to the Fortran compiler via the `FCFLAGS` variable (see above).

`SCALAPACK_LDFLAGS` Additional linker flags for ScaLAPACK usage.
Example:

```
SCALAPACK_LDFLAGS="-L$MKL_HOME/lib/intel64 \
  -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential \
  -lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm \
  -Wl,-rpath,$MKL_HOME/lib/intel64"
```

Note, that this variable is a convenience feature. You can also pass these flags to the linker via the `LDFLAGS` and `LIBS` variables (see above).

`PYTHON_CONFIG` Path to `python-config`.

`PYTHON_INCLUDE` Include flags for Python.

`NUMPY_INCLUDE` Include flags for NumPy.

Below in Sec. 3.2.3, we list and categorize the important options to configure ELPA. A full list of all available options can be obtained with `./configure --help`.

The configure options control which features are available at runtime. Whether a feature is actually used for the solution of the eigenproblem, depends on the ELPA settings chosen in your application. For example, if a specific kernel is enabled by a configuration option and it is not set as default, it must be activated with the `solver` setting (cf. Sec. 5.2.2) in order to be used for the computation.

3.2.2 Compiler flags for vectorization and optimization

In this section we give some hints on how to set the compiler vectorization and optimization flags, which are of the **utmost** importance for the ELPA performance.

Since (combinations of) these flags depend on the used compiler, its version, and the target hardware architecture, it is beyond the scope of this manual to give their comprehensive description and we refer further to the compiler/your HPC center documentation. In this section we still give some useful guides for these flags.

Vectorization The vectorization capabilities of the CPU should be fully exploited. Consult the documentation of your compiler to find the appropriate `FCFLAGS/CFLAGS/CXXFLAGS` for your system. It is important to enable the appropriate ELPA kernels (cf. Sec. 3.2.3) together with the correct compiler flags.
For GNU and Intel compilers, consider the `-march` and `-x` flags, respectively.
Examples: `FCFLAGS="-march=skylake-avx512 ..."` (GNU)
`FCFLAGS="-xCORE-AVX512 ..."` (Intel)

Optimization Compiler optimization should be switched on to enhance performance. You are adviced to select the highest optimization level that yields correct results. If the optimization is too aggressive, the calculated eigenvalues and eigenvectors are inaccurate. We do not generally recommend an optimization level, because this depends on the compiler and its version. Be aware that compiler vendors occasionally change the optimization strategies included in a certain level. Please always

	verify the correctness of your configuration with `make check` (see Sec. 3.3). Note that specifying the `-O` flag alone will typically *not* enable vectorization. **Examples:** For the GNU compilers (up to version 12) and the *classic* Intel compilers (`icc` and `ifort` up to Intel oneAPI version 2021.6), we recommend `-O2` or `-O3`. This is *not* valid for the new Intel `icx` and `ifx` compilers, which we have not yet tested thoroughly.
Threading	If ELPA should be run in hybrid parallelization with MPI and OpenMP, we recommend to link against the threaded math library. **Example:** `-lmkl_intel_thread` instead of `-lmkl_sequential` for Intel MKL
Floating-point calculations	To obtain accurate computational results, you should consider setting the flags for controlling the floating-point handling. **Example:** For the Intel compiler, consider changing the value of `-fp-model`. Setting `-fp-model=precise` will enable the most accurate calculations, however, with a potential performance penalty.

Some further useful hints can be found in the ELPA configuration examples, Sec. 3.2.4.

3.2.3 Configure options

In addition to the variables described in Sec. 3.2.1 the build of ELPA, can be controlled by adding options to the configure command line. Here, one has to distinguish

- standard configure options, offered by configure and
- ELPA specific configure options.

Listing here all the *standard configure options* is beyond the scope of this documentation. Most of these options are also only recommended for very experienced users. Concering the *ELPA specific configure options*, in this section we will focus here on most common ones, while other "expert" configure options are listed and explained in the Appendix A.

In any case, all configure options can be listed via the `./configure --help` command.

The general syntax for optional flags is `--enable-feature` or `--enable-feature=yes` for enabling the "feature" and `--disable-feature` or `--enable-feature=no` for disabling it. For some of the flags, the syntax is `--with-feature=yes` to enable and `--with-feature=no` to disable or `--with-feature=value` to specify a special flavor of a feature.

It is **strongly recommended** to always include the option `--enable-option-checking=fatal`, which aborts the configuration if any other option is unknown or invalid.

1. Controlling the installation directories

`--prefix`	Installation directory for architecture-independent files. Use this option if you want to install without root privileges. **Example:** `--prefix=$HOME/soft/elpa` **Default:** `/usr/local`
`--exec-prefix`	Installation directory for architecture-dependent files. **Default:** same as `--prefix` Further options are available for controlling the individual subdirectories. See `./configure --help` for a

detailed list.

2. Controlling the API provided by ELPA

   `--disable-skew-symmetric-support`
   Do not support skew-symmetric matrices. Removes `skew_eigenvectors`, `skew_eigenvalues`, etc. from the API.
   Default: `enabled`

   `--enable-python` Build and install the Python wrapper.
   Default: `disabled`

   `--disable-autotuning` Disable autotuning. See Sec. 6.1 for a detailed explanation.
   Default: `enabled`

3. Controlling MPI

   `--with-mpi=[yes|no]` Enable MPI parallelization.
   Default: `yes`
   Note that MPI parallelisation should only be switched off for very good reasons and that ELPA execution is then limitied to one compute node!

   `--disable-mpi-module` Replace the Fortran MPI module `use mpi` with `include "mpif.h"`. Use this option only if your MPI library does not provide a Fortran module. Typically error messages are "no module mpi" or "cannot open module mpi".
   Default: `enabled` (= use mpi module)

   `--disable-detect-mpi-launcher`
   Disable automatic detection of MPI launcher.
   Default: `enabled`

   `--enable-mpi-launcher=[mpiexec|mpiexec.hydra|mpirun|srun]`
   Use specified MPI launcher. for running a `make check` test suit in HPC systems that don't allow interactive MPI runs. E.g. using a combination `"--disable-detect-mpi-launcher --enable-mpi-launcher=srun"` one can call a `make check` from a SLURM script on a system that supports `srun`.

4. Controlling OpenMP

   `--enable-openmp` Compile with OpenMP threading parallelism. Note that independent of whether ELPA has been built with threading support, you can always use multi-threading for your math library if ELPA is properly linked against its threaded version. See also Sec. 5.5.
   Default: `disabled`

5. Availability of ELPA2 compute kernels

Note that at the end of the configuration, a list of all enabled kernels will be displayed.

`--disable-generic-kernels` Do not build generic kernels compatible with all platforms. Note that the performance of these kernels will be inferior to other vectorized kernels.
Default: `enabled`

`--disable-sse-kernels` Do not build SSE kernels.
Default: `enabled`

`--disable-sse-assembly-kernels`
Do not build SSE kernels written in assembly.
Default: `enabled`

`--disable-avx-kernels` Do not build AVX kernels for Intel Sandy Bridge and later.
Default: `enabled`

`--disable-avx2-kernels` Do not build AVX2 kernels for Intel Haswell and later.
Default: `enabled`

`--disable-avx512-kernels` Do not build AVX-512 kernels for Intel Knights Landing and later.
Default: `enabled`

`--enable-vsx-kernels` Build VSX kernels for IBM POWER7 and later.
Default: `disabled`

`--enable-sparc64-kernels` Build kernels for processors supporting SPARC64 (SPARC V9).
Default: `disabled`

`--enable-bgp-kernels` Build kernels for IBM Blue Gene/P.
Default: `disabled`

`--enable-bgq-kernels` Build kernels for IBM Blue Gene/Q.
Default: `disabled`

`--enable-neon-arch64-kernels`
Build kernels for ARM using Neon (Advanced SIMD) instructions.
Default: `disabled`

`--enable-sve128-kernels` Build 128-bit SVE kernels for ARM processors.
Default: `disabled`

`--enable-sve256-kernels` Build 256-bit SVE kernels for ARM processors.
Default: `disabled`

`--enable-sve512-kernels` Build 512-bit SVE kernels for ARM processors.
Default: `disabled`

`--with-fixed-real-kernel=KERNEL`
Build only a single specific real kernel and make it default. Avialable kernels are: `generic`, `generic_simple`, `generic_simple_block4`, `generic_simple_block6`, `sparc64_block2`,

sparc64_block4, sparc64_block6, neon_arch64_block2, neon_arch64_block4, neon_arch64_block6, vsx_block2, vsx_block4, vsx_block6, sse_block2, sse_block4, sse_block6, sse_assembly, sve128_block2, sve128_block4, sve128_block6, avx_block2, avx_block4, avx_block6, avx2_block2, avx2_block4, avx2_block6, sve256_block2, sve256_block4, sve256_block6, avx512_block2, avx512_block4, avx512_block6, sve512_block2, sve512_block4, sve512_block6, bgp, bgq, nvidia_gpu, amd_gpu, intel_gpu_sycl, nvidia_sm80_gpu.

`--with-fixed-complex-kernel=KERNEL`
: Build only a single specific complex kernel and make it default. Avialable kernels are: generic, generic_simple, neon_arch64_block1, neon_arch64_block2, sse_block1, sse_block2, sse_assembly, sve128_block1, sve128_block2, avx_block1, avx_block2, avx2_block1, avx2_block2, sve256_block1, sve256_block2, avx512_block1, avx512_block2, sve512_block1, sve512_block2, bgp, bgq, nvidia_gpu, amd_gpu, intel_gpu_sycl, nvidia_sm80_gpu.

`--with-default-real-kernel=KERNEL`
: Set a specific real kernel as default. See `--with-fixed-real-kernel` for a complete list of avaliable kernels.
Default: `real_avx512_block2`

`--with-default-complex-kernel=KERNEL`
: Set a specific complex kernel as default. See `--with-fixed-complex-kernel` for a complete list of avaliable kernels.
Default: `complex_avx512_block1`

`--enable-heterogenous-cluster-support`
: **Experimental!** Select a kernel supported by all CPUs in a heterogenous cluster. Currently, only available for Intel CPUs.
Default: `disabled`

6. Controlling the AMD GPU version
The build of the AMD GPU version can be controlled by

`--enable-amd-gpu-kernels`
: Build kernels for AMD GPUs.
Default: `disabled`

If this option is enabled, then the details of the AMD GPU version can be further controlled with

`--enable-gpu-streams=[nvidia|amd|no]`
: Use Cuda or HIP streams in Nvidia or AMD GPU versions, respectively.

Default: `no` (disabled)

`--with-AMD-gpu-support-only=[yes|no]`
: **Experimental!** Build real and complex AMD GPU kernels only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-rocsolver=[yes|no]`
: Use AMD rocSOLVER library.
Default: `no`

`--enable-marshalling-hipblas-library`
: Use indirection layer hipBLAS instead of rocBLAS.
Default: `disabled`

`--enable-hipcub`
: Use reductions from hipCUB in AMD GPU kernels.

`--enable-gpu-ccl=[nccl|rccl|no]`
: Use NCCL or RCCL communication libraries in Nvidia or AMD GPU versions, respectively.
Default: `no` (disabled)

7. Controlling the Intel GPU version
The build of the AMD GPU version can be controlled by

`--enable-intel-gpu-sycl-kernels`
: Build kernels for Intel GPUs using SYCL. Requires `--enable-intel-gpu-backend=sycl`.
Default: `disabled`

`--enable-intel-gpu-backend=[sycl|openmp]`
: Build GPU code for Intel GPUs and select either SYCL or OpenMP as the backend.
Default: `disabled` (= no Intel GPU kernels)

If this options are set/enabled to use SYCL then one can further controll the build with

`--with-INTEL-gpu-support-only=[yes|no]`
: **Experimental!** Build real and complex Intel GPU kernels only. If enabled, no other kernels will be available at runtime.
Default: `no`

8. Controlling the Nvidia GPU version
In the past, when only a Nvidia GPU version was available, Nvidia GPU buils were triggered by

`--enable-gpu`
: **Deprecated.** Build kernels for GPUs. Please use explicit options for the various vendors instead.
Default: `disabled`

This configure argument is outdated and **will be removed in one of the next releases. Do not use it anymore!**

Instead, nowadays, the Nvidia GPU build must be enabled with one of the two following arguments must be used:

`--enable-nvidia-gpu-kernels`
: Build kernels for Nvidia GPUs. Use `--with-NVIDIA-GPU-compute-capability` to set the compute capability for best performance.
Default: `disabled`

`--enable-nvidia-sm80-gpu-kernels`
: Build kernels for Nvidia GPUs supporting the compute capability 8.0, for example, Nvidia A100.
Default: `disabled`

If the Nvidia GPU build is enabled, then it can be further controlled with the arguments:

`--with-cuda-path=PATH`
: Path where CUDA is installed.
Default: detect automatically

`--enable-cuda-aware-mpi`
: Use CUDA-aware MPI features to enhance performance. Requires an MPI library that integrates with CUDA, for example, OpenMPI.
Default: `disabled`

`--enable-gpu-streams=[nvidia|amd|no]`
: Use CUDA or HIP streams in Nvidia or AMD GPU versions, respectively.
Default: `no` (disabled)

`--enable-gpu-ccl=[nccl|rccl|no]`
: Use NCCL or RCCL communication libraries for Nvidia or AMD GPU versions, respectively.
Default: `no` (disabled)

`--with-nccl-path=PATH`
: Path where NCCL is installed.
Default: detect automatically

`--with-NVIDIA-gpu-support-only=[yes|no]`
: Build real and complex Nvidia GPU kernels only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-NVIDIA-sm_80-gpu-support-only=[yes|no]`
: Build real and complex Nvidia GPU kernels for compute capability 8.0 only. If enabled, no other kernels will be available at runtime.
Default: `no`

`--with-NVIDIA-GPU-compute-capability=VALUE`
: Use compute capability `VALUE` for Nvidia GPU kernels.
Default: `sm_35` (= 3.5)

`--enable-NVIDIA-gpu-memory-debug`
: Output memory information of Nvidia GPU devices. The script at `utils/memory/check\_memory.py` can be used to process the output.
Default: `disabled`

`--with-cusolver=[yes|no]`
: Use Nvidia cuSolver library.
Default: `no`

`--enable-nvidia-cub` Use reductions from CUB in real Nvidia GPU kernel.
Default: `disabled`

`--enable-nvtx` Build and install NVTX wrappers for profiling the GPU code.
Default: `disabled`

9. Controlling performance related options

`--disable-autotuning` Disable autotuning. See Sec. 6.1 for a detailed explanation.
Default: `enabled`

`--with-papi=[yes|no]` Use PAPI to measure and print FLOP counts. Only available if `--enable-timings` is set.
Default: `no`

`--with-likwid=[yes|no|PATH]`
Use LIKWID to measure the performance of some solver parts. If set to `yes` and the library can not be found, you can set the `PATH` explicitly.
Default: `no`

`--disable-assumed-size` Do not use assumed-size Fortran arrays.
Default: `enabled`

10. Output

`--disable-timings` Disable timings measurement with the API functions `timer_start`, `timer_stop`, `get_time`, and the output of `print_times`. If disabled, some ELPA features like autotuning will not work.
Default: `enabled`

`--enable-redirect` **For test programs.** Redirect stdout and stderr of each MPI task to a separate file in the subdirectory `mpi_stdout`.
Default: `disabled`

11. Precision

`--enable-single-precision` Use single precision in addition to double precision.
Default: `disabled` (= double precision)

`--enable-64bit-integer-math-support`
Support 64-bit integers in the math libraries BLAS, LAPACK, and ScaLAPACK. Combine this option only with the appropriate link line to the math library, e.g., by choosing the suffix `_ilp64` for Intel MKL.
Default: `disabled`

`--enable-64bit-integer-mpi-support`
Support 64-bit integers in the MPI library. Make sure to link against the appropriate MPI library.
Default: `disabled`

12. Fortran

`--disable-Fortran2008-features`
: Do not use Fortran 2008 features. Use this option if your compiler does not support the Fortran 2008 standard.
Default: `enabled`

`--enable-ifx-compiler`
: Compile with `ifx`.
Default: `disabled`

13. Controlling which test programs will be build

`--enable-c-tests`
: Build C tests.
Default: `enabled`

`--enable-cpp-tests`
: Build C++ tests.
Default: `enabled`

`--enable-scalapack-tests`
: Build ScaLAPACK test cases for performance comparison.
Default: `disabled`

`--enable-python-tests`
: Enable Python tests. Only available if `--enable-python` is set.
Default: `disabled`

3.2.4 `configure` examples

The following examples should provide an overview of how to configure ELPA. They are, however, not meant for a production-ready build. For the best performance, they have to be optimized for the respective system.

- **OpenMP, GNU compilers**

To configure a threaded build without MPI support on your personal linux workstation, you can try using this command ("\" symbol is used for a line break and can be omitted):

```
../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal CC=gcc \
CXX=g++ FC=gfortran CFLAGS="-O3 -march=native" FCFLAGS="-O3 -march=native" \
--disable-avx512 --with-mpi=no --enable-openmp
```

We assume here that you have the current GNU compiler suite and all required libraries installed in the default location. These requirements are often met on the well-known linux distributions. If the math libraries can not be found automatically, you need to explicitly set the variables `SCALAPACK_FCFLAGS` and `SCALAPACK_LDFLAGS` (see Sec. 3.2.1) and other examples below.

- **MPI+OpenMP, GNU compilers**

```
../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal CC=mpicc \
CXX=mpicxx FC=mpifort CFLAGS="-O3 -march=native" FCFLAGS="-O3 -march=native" \
--disable-avx512 --with-mpi=yes --enable-openmp
```

- **MPI, Intel toolchain**

Here we present an example of a configure line for a system with an Intel CPU that doesn't support AVX-512 instructions. We are using the Intel classic compilers in combination with Intel MKL and Intel MPI ("Intel toolchain"):

```
../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal CC=mpiicc \
```

```
CXX=mpiicpc FC=mpiifort CFLAGS="-O3 -xHost" FCFLAGS="-O3 -xHost" \
SCALAPACK_FCFLAGS="-I${MKLROOT}/include/intel64/lp64" \
SCALAPACK_LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_scalapack_lp64 \
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 \
-lpthread -lm -Wl,-rpath,${MKLROOT}/lib/intel64" \
--with-mpi=yes --disable-avx512
```

Note that `-lmkl_intel_thread` should be used instead of `-lmkl_sequential` for the threaded build. and also that `-lmkl_blacs_openmpi_lp64` should be used instead of `-lmkl_blacs_intelmpi_lp64` if you use Open MPI instead of Intel MPI. For the further details specific to Intel MKL we refer to Intel MKL Link Line Advisor [1].

At the time of writing, all necessary Intel tools including `icc/ifort` compilers, Intel MKL, and Intel MPI are available free of charge as a part of Intel oneAPI HPC Toolkit.

- **Nvidia GPU + MPI, Intel toolchain**

If you want to use ELPA on a Nvidia GPU-accelerated system with Intel CPU supporting AVX-512 (Intel Knights Landing and later) and using Intel toolchain:

```
../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal \
CC=mpiicc FC=mpiifort CXX=mpiicpc \
CFLAGS="-O3 -xCORE-AVX512 -I$MKLROOT/include/intel64/lp64 \
  -I$CUDA_HOME/include" \
FCFLAGS="-O3 -xCORE-AVX512 -I$MKLROOT/include/intel64/lp64
   -I$CUDA_HOME/include" \
LDFLAGS="-L$MKLROOT/lib/intel64 -lmkl_scalapack_lp64 -lmkl_intel_lp64 \
-lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm \
-Wl,-rpath,$MKLROOT/lib/intel64" \
--with-mpi=yes --enable-nvidia-gpu-kernels \
--with-NVIDIA-GPU-compute-capability=sm_80 --with-cuda-path=$CUDA_HOME \
```

Here `$CUDA_HOME` is the path to the CUDA installation directory.

- **Intel GPU + MPI, Intel toolchain**

If you want to use ELPA on a Intel GPU-accelerated system on top of the Intel oneAPI toolchain:

```
../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal
   FC="mpiifort -fc=ifx" CC=mpiicx CXX=mpiicpx CFLAGS="-O3 -xCORE-AVX512"
   CXXFLAGS="-O3 -xCORE-AVX512 -fsycl
   -I$ONEAPI_ROOT/compiler/latest/linux/include/sycl
   -I$ONEAPI_ROOT/mkl/latest/include" FCFLAGS="-O3 -xCORE-AVX512 -fsycl
   -I$ONEAPI_ROOT/compiler/latest/linux/include/sycl"
   LIBS="-L$ONEAPI_ROOT/compiler/latest/linux/lib
   -L$ONEAPI_ROOT/compiler/latest/linux/compiler/lib/intel64_lin -lsycl
   -Wl,-rpath,$ONEAPI_ROOT/compiler/latest/linux/lib"
   SCALAPACK_FCFLAGS="-I$ONEAPI_ROOT/mkl/latest/include/intel64/lp64 -fsycl"
   SCALAPACK_LDFLAGS="-fsycl -L$ONEAPI_ROOT/mkl/latest/lib/intel64 -lmkl_sycl
   -lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core
   -lmkl_blacs_intelmpi_lp64 -lsycl -lOpenCL -lpthread -lm -ldl -lirng -lstdc++
   -Wl,-rpath,$ONEAPI_ROOT/mkl/latest/lib/intel64" --enable-ifx-compiler
   --with-mpi=yes --enable-sse-kernels --enable-sse-assembly-kernels
```

[1] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html

```
--enable-avx-kernels --enable-avx2-kernels --enable-avx512
--enable-intel-gpu-sycl-kernels --enable-intel-gpu-backend=sycl
--enable-single-precision
```

TODO!!! Add some more examples for configure lines? Which ones?

- **AMD GPU ?**

3.3 Building

After the successful configuration with the appropriate options for your system, ELPA can be built with `make`. Depending on your machine, you can speed up this process with the command line argument `-j` followed by the number of cores to use for building.

When ELPA has been compiled and linked successfully, we recommend running the included test suite with `make check`. It supports the following options:

`CHECK_LEVEL`	If set to `extended`, run additional time-consuming tests. If set to `autotune`, run additional tests for verifying the autotuning feature of ELPA. **Default:** Run the basic test suite.
`TASKS`	Number of MPI tasks to use for testing. **Default:** 2
`TEST_FLAGS`	Tuple of matrix size, number of eigenvalues, and block size. **Example:** `TEST_FLAGS="150 100 32"` for a matrix of 150×150 requesting 100 eigenvalues using a block size of 32. Smaller matrices speed up the '`make check`' test suite. **Default:** `"5000 150 16"`

3.4 Complete installation example

Here we present an example of a complete installation of ELPA for a linux workstation with Intel CPU, using the Intel toolchain.

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa
./autogen.sh

mkdir build
cd build

../configure --prefix=$HOME/soft/elpa --enable-option-checking=fatal \
CC=mpiicc CXX=mpiicpc FC=mpiifort CFLAGS="-O3 -xHost" FCFLAGS="-O3 -xHost" \
SCALAPACK_FCFLAGS="-I${MKLROOT}/include/intel64/lp64" \
SCALAPACK_LDFLAGS="-L${MKLROOT}/lib/intel64 -lmkl_scalapack_lp64 \
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 \
-lpthread -lm -Wl,-rpath,${MKLROOT}/lib/intel64" \
--with-mpi=yes --disable-avx512

make -j 8
make check -j 8 TEST_FLAGS="150 100 32"
make install
```

If the installation was successful, ELPA is now installed in the directory `$HOME/soft/elpa` and you'll get a message how to link your application against ELPA. More details on compiling and linking against ELPA can be found in Sec. 4.

If the installation was not successful, we give some (although incomprehensive) hints on troubleshooting in the following section.

3.5 Troubleshooting

Typically errors can during one of these stepps:

- `configure`
- `make`
- `make check`

If the error occured during `make` or `make check`, make sure to clear the build directory before re-running `make` or `make check` before reconfiguring ELPA with new flags.

3.5.1 `configure` problems

Most typical errors at the `configure` stage are related to missing dependencies. Please make sure that you have installed all required dependencies (see Sec. 3.1) and carefully read the error message, since it can give a hint which dependency is missing. For the extended error message, check the `config.log` file in the build directory.

Problem. If you use the GNU compilers and encounter the error `initializer element is not constant` or `not specified in enclosing 'parallel'` during build, this is most likely caused by an outdated gcc compiler version.

Solution. Make sure that you use gcc of at least version 10.

3.5.2 `make` problems

Problem. There is an error message related to MPI, e.g. `"no module mpi"`, `"cannot open module mpi"`, `"Could not resolve generic procedure mpi_send/mpi_recv/mpi_allreduce"`.

Solution. Try to reconfigure ELPA with the option `--disable-mpi-module`. It doesn't switch off MPI, it just affects internal working of ELPA, so that it does not use the Fortran MPI module, but rather get interfaces by "include mpif.h".

3.5.3 `make check` problems

Extended error messages can be found in the `test-suite.log` file in the build directory. Note that the Fortran tests, e.g. `validate_complex_double_eigenvectors_...` are usually more expressive in describing the error than the corresponding C/C++ tests, e.g.
`validate_c_version_complex_double_eigenvectors_.../`
`validate_cpp_version_complex_double_eigenvectors_...`

Problem. Some HPC centers do not allow running MPI programs interactively. It therefore could happen that `make check` does not run at all on the machine on which you are installing ELPA.

Solution. If the HPC center supports SLURM with `srun`, you can reconfigure ELPA with the following options: `--disable-detect-mpi-launcher --enable-mpi-launcher=srun` (see

Sec. 3.2.3) and then call `make check` from a SLURM script. Alternatively, consult the documentation of your HPC center on how to *interactively* run MPI programs.

Problem. `"Program Exception - illegal instruction"` error and/or errors in `compute_hh_trafo`.

Solution. Try to reconfigure ELPA with disabled vectorization options, e.g. `--disable-avx512`.

Problem. `make check` is too slow.

Solution. Use the `-j` option for utilizing more cores and A smaller text matrix size to speed up the tests, e.g. `make check -j 8 TEST_FLAGS="150 100 32"` for using 8 cores and the matrix of size 150×150. You can also switch off certain ELPA APIs and tests by reconfiguring ELPA with the flags `"--disable-skew-symmetric-support --disable-c-tests --disable-cpp-tests"`.

4 Compiling and linking against ELPA

To link your application against your local installation of the ELPA library, you need to point the compiler to the correct include files (header file for C/C++ or module file for Fortran) and instruct the linker to find the library file.

4.1 Linking with `pkg-config`

The best option is to use the *package config tool*. Make sure to install the program `pkg-config` on your system. The following steps explain how to fetch the correct flags. Note that you usually forward them to the compiler and linker as `FCFLAGS`, `CFLAGS`, `CXXFLAGS`, and `LDFLAGS`.

1. Extend the `PKG_CONFIG_PATH` environment variable to point to the subfolder `lib/pkgconfig` (or `lib64/pkgconfig` on some systems) of your ELPA installation. Depending on your system and shell, this *might* look similar to this:

   ```
   export
   PKG_CONFIG_PATH="/absolute_path_to_elpa/lib/pkgconfig:$PKG_CONFIG_PATH"
   ```

   where `/absolute_path_to_elpa` corresponds to the absolute path of the ELPA installation set by `--prefix` (e.g. `$HOME/soft/elpa` from Sec. 3.2.4)

2. To fetch the correct flags for Fortran (`FCFLAGS`), run the command

   `pkg-config --variable=fcflags elpa`

   or

   `pkg-config --variable=fcflags elpa_openmp`

   depending on your build.

3. To fetch the correct flags for C/C++ (`CFLAGS` or `CXXFLAGS`), run the command

   `pkg-config --cflags elpa`

   or

   `pkg-config --cflags elpa_openmp`

   depending on your build.

4. To fetch the correct linker flags (`LDFLAGS`), run the command

   `pkg-config --libs elpa`

   or

   `pkg-config --libs elpa_openmp`

   depending on your build.

Adding these flags to the build procedure of your application will link it against ELPA. It should be mentioned that these flags will include all necessary options for libraries that ELPA has been linked against during its build, especially the GPU, MPI, BLACS, BLAS, LAPACK, and ScaLAPACK libraries. If your application relies also on one or more of these libraries, the linkline is “shipped” with the ELPA linkline and explicit linking might not be necessary.

4.2 Linking without `pkg-config`

If you do not want to use the `pkg-config` tool, although we strongly recommend doing that, you can also set the flags manually. For most compilers, the include flags (added to `FCFLAGS`, `CFLAGS`, or `CXXFLAGS`) should be

```
-I/absolute_path_to_elpa/include/build_specific_subdirectory
```

where, as before, `/absolute_path_to_elpa` corresponds to the absolute path of the ELPA installation set by `--prefix` (e.g. `$HOME/soft/elpa` from Sec. 3.2.4) and `/build_specific_subdirectory` is something like `elpa-2023.05.001`

The linker flags (`LDFLAGS`) are typically

```
-L/absolute_path_to_elpa/lib -lelpa
```

or

```
-L/absolute_path_to_elpa/lib -lelpa_openmp
```

depending on your build. Make sure that you adapt the paths and flags accordingly. Note that unlike in the case of `pkg-config --libs`, here the `LDFLAGS` do not automatically contain links to external libraries (MPI, BLACS, etc.).

It might happen at *runtime* that the ELPA library cannot be found. In this case either set the `LD_LIBRARY_PATH` pointing to the ELPA library directory with (depending on your system and shell)

```
export LD_LIBRARY_PATH="/absolute_path_to_elpa/lib:$LD_LIBRARY_PATH"
```

or add an additional linker flag

```
-Wl,-rpath,/absolute_path_to_elpa/lib
```

to the `LDFLAGS` when building your application. In the latter case, setting the `LD_LIBRARY_PATH` is not necessary anymore.

5 Calling ELPA

In this section, the ELPA Fortran API is explained first followed by illustrations of the steps involved to setup ELPA and use it from within an application code. For guidlines on using the Python API, please see Sec. 5.9.

5.1 API version

Each ELPA release defines two version numbers for the API. First, the *release API version*, for the latest release also often referred to as *current API version*, and the *minimal API version* supported by this release. Obviously, the versioning scheme of ELPA API versions is monotonistic growing such that a natural ordering (lower API version means older) can be infered.

The *minimal API version* tells you whether there have been breaking changes in the API, i.e. whether downward compatibility only to a certain ELPA release (identified by the *release API version* of this old release being the same as the *minimal API version* of the newer release) is guranteed. Up to now this has been never happening for the ELPA library, but might potentially occur in the future.

A change in the *release API version*, implies that there have either been changes to the API or whether new *key-value pairs* (see Sec. 5.2) have been introduced. Typically, the *release API version* is increased if new procedures have been *added* with a release. If the *minimal API version* did not change from one release to the other, it also implies that nothing has been *removed* from the API. As mentioned, the API version of a new release will also be changed if new *key-value pairs* have been introduced, to allow for new funtionality or performance tuning. It is important to note that adding new key-value pairs does not introduce breaking changes, since an application making use of these new key-value pairs can be still linked against and run with older ELPA versions not supporting these keywords. The only change will be that the older ELPA library will ignore the new keywords but still run and produce correct results, albeit with maybe lower performance than a newer ELPA release. Even removing key-value pairs would only introduce a "breaking change" insofar that the key-value combination would be ignored and performance might drop, but again, ELPA would continue to work and produce the correct results.

Nevertheless, it is recommended to upgrade your application to the latest versions of ELPA availabe and to initialize with the latest *release API version* since only this does gurantee you to obtain the best possible performance from the ELPA library.

An overview over the EPLA versions published and with the repective *release API version* and *minimal API version* is shown in the Table 1. Note that before the release of ELPA 2017.05.001 another API has been used and breaking API changes occured with every release. With the introduction of the API of release 2017.05.001 the API become much more expressive and stable.

For a given ELPA installation you can find out the supported API versions by either referring to the Table 1, or by inspecting the file `elpa_version.h` in your ELPA installation path.

5.2 Key-Value pairs

Every ELPA object is controlled via key-value pairs. Note, that ELPA knows two types of key-value pairs:

- **Mandatory parameters**: settings which are *fixed* for the lifetime of an ELPA object and *must* be set *before* calling the `setup` procedure, e.g. a matrix size. If you want to change any of these parameters, you have to create a new ELPA object. Note that as many ELPA

ELPA release	Release API version	Minimal supported API vesion
2023.05.001	20231705	20170403
2022.11.001	20221109	20170403
2022.05.001	20211125	20170403
2021.11.001	20211125	20170403
2020.11.001	20190501	20170403
2020.05.001	20190501	20170403
2020.05.001	20190501	20170403
2019.11.001	20190501	20170403
2019.05.001	20190501	20170403
2018.11.001	20181113	20170403
2018.05.001	20180525	20170403
2017.11.001	20171201	20170403
2017.05.001	20170403	20170403

Table 1: ELPA release versions and the corresponding API versions

objects as needed can be instanciated at the same time. These parameters are listed in Sec. 5.2.1.

- **Runtime options**: key-value pairs which control the runtime of the ELPA library for a given ELPA object. These options might either control the program flow, such as using GPUs or the 1-stage or 2-stage solver, or the performance of the ELPA library, by tuning the algorithmic execution to the hardware and problem size. Whether a key-value pair is available or has an effect, depends on the supported API version of the ELPA library used (see Sec. 5.1), the API version initialized, and also on the build options of the ELPA-library. Runtime option values can be adjusted between calls to the ELPA math-routines. Most common runtime options are listed in Sec. 5.2.2, and some additional expert options are listed in Appendix. B.

The values of key-value pairs can be integers, floating-point (float or double) numbers, boolean flags (`0` or `1`), or special data types. The accepted values are specified below together with their default values if applicable.

5.2.1 Mandatory parameters

The following key-value pairs are mandatory parameters which must be set for each ELPA object, **before** calling the `setup` procedure and then cannot be changed anymore:

`na` — *Integer.* Global matrix size $\texttt{na} \times \texttt{na}$.

`nev` — *Integer.* Number of eigenvalues and/or eigenvectors to be computed. $0 \leq \texttt{nev} \leq \texttt{na}$

`nblk` — *Integer.* Block size for the block-cyclic matrix layout. Must be a power of two. Typical values for CPU execution are 16 or 32. For GPU computations, 64 or larger are favorable and 1024 is the maximal allowed value. Note that the parameter's value should be chosen in accordance with the the most favorable BLACS distribution of your application.

`local_nrows` — *Integer.* Number of rows of the local matrix stored on the given MPI process. Can be determined using the ScaLAPACK function `numroc`.

`local_ncols` — *Integer.* Number of columns of the local matrix stored on the given

	MPI process. Can be determined using the ScaLAPACK function `numroc`.
`mpi_comm_parent`	*MPI_Comm.* Global MPI communicator comprising all MPI ranks used by ELPA. Mandatory if MPI is enabled.
`blacs_context`	*Integer.* The blacs context for the valid BLACS distribution as obtained from the BLACS funtions.

Note that it is mandatory the set the parameters `local_nrows`, `local_ncols` to describe the dimension of the local sub-matrices of the distributed global matrix of size $na \times na$. It is also mandatory to set the parameter `mpi_comm_parent` to provide the global MPI communicator of all ranks to be used in the calculations.

However, ELPA does also need the information how the MPI setup is spanning a 2D grid of row and column MPI processes. You have two possible ways how to provide this information to ELPA:

1. The splitting of the `mpi_comm_parent` (typically that is `MPI_COMM_WORLD`) into the `mpi_comm_rows` and `mpi_comm_cols` communicators is done in your application *before* the ELPA object is setup. Then you can provide this communicators to ELPA. If you choose this option it is mandatory to set the following parameters:

`mpi_comm_rows`	*MPI_Comm.* MPI communicator for the MPI processes organized in rows.
`mpi_comm_cols`	*MPI_Comm.* MPI communicator for the MPI processes organized in columns.

2. ELPA should internally split the provided `mpi_comm_parent` communicator into the internally used `mpi_comm_rows` and `mpi_comm_cols` communicators. If you choose this option since you do not want to provide the `mpi_comm_rows` and `mpi_comm_cols` communicators, it is mandatory to set the following parameters:

`process_row`	*Integer.* The row id of the MPI rank in the row communicator as obtained from the BLACS routines.
`process_col`	*Integer.* The column id of the MPI rank in the column communicator as obtained from the BLACS routines.

Note that per instanciated ELPA object one has to decide for **one** of the two options discussed above. It is not allowed to provide a combination of the parameters from both options, since the `setup` method will not accept such input.

In addition to the above mentioned mandatory parameters for seting up the ELPA object, one can provide *additional* parameters to describe the MPI setup:

`num_processes`	*Integer.* Total number of MPI ranks in `mpi_comm_parent`.
`num_process_rows`	*Integer.* Total number of MPI ranks in `mpi_comm_rows`.
`num_process_cols`	*Integer.* Total number of MPI ranks in `mpi_comm_cols`.
`process_id`	*Integer.* Rank number of each MPI task in `mpi_comm_parent`.

Setting these parameters is not necessary, since ELPA can deduce them from the mandatory parameters and will set them internally if they are not provided by the user. However, it is recommended to set them, since we have observed that this helps users to organize their code and keep an understanding on how the ELPA object is set up.

5.2.2 Runtime options

The following parameters are optional.

5.2.2.a General runtime options

`omp_threads`	*Integer.* Number of OpenMP threads to use. Only relevant if ELPA has been configured with `--enable-openmp`. **Default:** `1`
`solver`	Either `ELPA_SOLVER_1STAGE` or `ELPA_SOLVER_2STAGE`. Specify which solver to use: ELPA1 or ELPA2. This choice can influence the performance considerably. If unsure, measure and compare the runtime of both solvers. **Default:** `ELPA_SOLVER_1STAGE`
`real_kernel`	Real kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`. **Default:** set by configuration option `--with-default-real-kernel`
`complex_kernel`	Complex kernel to use if `solver` is set to `ELPA_SOLVER_2STAGE`. **Default:** set by configuration option `--with-default-complex-kernel`

5.2.2.b Runtime options for GPU

The following parameters are related to running ELPA on GPUs. All flags can be enabled or disabled by setting them to `1` or `0`, respectively.

`nvidia-gpu`	Enable GPU acceleration using Nvidia GPUs. Only available if ELPA has been configured with `--enable-nvidia-gpu`. **Default:** `0` (= disabled)
`intel-gpu`	Enable GPU acceleration using Intel GPUs. Only available if ELPA has been configured with `--enable-intel-gpu-backend=sycl` and `--enable-intel-gpu-sycl-kernels`. Additionally, ELPA must be configured with the support of Intel ifx compiler `--enable-ifx-compiler`. **Default:** `0` (= disabled)
`amd-gpu`	Enable GPU acceleration using AMD GPUs. Only available if ELPA has been configured with `--enable-amd-gpu`. **Default:** `0` (= disabled)
`use_gpu_id`	*Integer.* Specify which GPU should be used by the calling MPI task.

5.2.2.c Runtime options for debugging

The following switches control additional measurements or output, which can be conveniently used for debugging.

`verbose`	Print verbose information about calculations and errors. This option can be enabled without performance loss. **Default:** `0` (= disabled)
`debug`	Print debugging information. Additional checks are performed and additional timing information is gathered. Enabling this option decreases performance and is not recommended for production.

Default: `0` (= disabled)

`output_build_config`
Print the options with which ELPA has been configured and built.

`output_pinning_information`
Print pinning information, i.e. association of OpenMP threads to cores.
Default: `0` (= disabled)

`print_flops`
Enable printing FLOP rates.
Default: `0` (= disabled)

`timings`
Enable the detailed timings measurement. Only available if ELPA has been configured with `--enable-timings`. This option can be enabled without performance loss. It shouldn't be disabled if autotuning is used.
Default: `1` (= enabled)

`measure_performance`
Measure FLOP rates together with the timings using PAPI. Only available if ELPA has been configured with `--enable-timings`.
Default: `0` (= disabled)

5.3 Math routines provided by ELPA

ELPA provides numerous math routines needed for solving symmetric, or hermitian (generalized) eigenvalue problems. If not stated otherwise, all routines are available for real and complex double-precision calculations. If ELPA has been build with single-precision support, the routines are also available for real and complex single-precision datatypes.

In the following "all datatypes" means real and complex double and single-precision. Please note that all ELPA procedures have in common that they have a sligthly different synopsis depending whether ELPA is used form Fortran or C/C++. The difference, however, follows a single pattern:

- In Fortran programs ELPA procedures are always used in the form `yourELPAobject%procedurename`.
- In C/C++ programs ELPA procedures have always an *additional first argument* – the `handle` to `yourELPAobject` and procedure names are preceded with the prefix `elpa_`.

For simplicity, only the Fortran synopsis is shown here. More details, and also the C/C++ synopsis can be found in Appendix C.

5.3.1 Standard eigenvalue problem

Important note for the GPU users. The overloaded convenience functions, like `eigenvalues()` can only be used if the data has been allocated on the host. If the data has been allocated on the GPU device, an automatic destinction of datatypes is not possible and one has to use the explicit functions specifying the datatype, e.g. `eigenvalues_double()`.

For the *standard* eigenvalue problem the following routines are provided:

`TODO!!!` Add datatypes for the parameters? Or just directly convert man to pdf and put to appendix?

`eigenvalues(a, ev, error)`
: Overloaded function (for all datatypes) that returns only the eigenvalues. Here:
`a` is the *host* matrix,
`ev` is the *host* eigenvalue array,
`error` is the return code.

`eigenvectors(a, ev, z, error)`
: Overloaded function (for all datatypes) that returns (part of) the eigenvalues and the corresponding eigenvectors. Here:
`a` is the *host* matrix,
`ev` is the *host* eigenvalue array,
`z` is the *host* matrix of eigenvectors,
`error` is the return code.

`eigenvalues_[double|single|complex_double|complex_single](a, ev, error)`
: Explicit function (for all datatypes) that returns only the eigenvalues. Here:
`a` is the *host/device* matrix,
`ev` is the *host/device* eigenvalue array,
`error` is the return code.

`eigenvectors_[double|single|complex_double|complex_single](a, ev, z, error)`
: Explicit function (for all datatypes) that returns (part of) the eigenvalues and the corresponding eigenvectors. Here:
`a` is the *host/device* matrix,
`ev` is the *host/device* eigenvalue array,
`z` is the *host/device* matrix of eigenvectors,
`error` is the return code.

Note that if ELPA has been build with the support for real skew-symmetric matrices, then in addition the procedures `skew_eigenvalues`, `skew_eigenvalues_[double|float]`, `skew_eigenvectors` and `skew_eigenvectors_[double|float]` are available.

5.3.2 Generalized eigenvalue problem

Important note for the GPU users. There are currently no routines for the generalized eigenvalue problems that support that the data is already allocated on a GPU. This will come in the next release.

For the *generalized* eigenvalue problem $AZ = \lambda BZ$ the following routines are provided:

`generalized_eigenvalues(a, b, ev, isAlreadyDecomposed, error)`
: Overloaded function (for all datatypes) that only returns (part of) the eigenvalues. Here
`a` is the *host* matrix,
`b` is the B *host* matrix,
`ev` is the *host* eigenvalue array,
`z` is the *host* matrix of eigenvectors,
`error` is the return code.
`isAlreadyDecomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls.

`generalized_eigenvectors(a, ev, b, z, isAlreadyDecomposed, error)`
: Overloaded function (for all datatypes) that returns (part of) the eigenvalues and the corresponding eigenvectors. Here
`a` is the *host* matrix,

`b` is the B *host* matrix,
`ev` is the *host* eigenvalue array,
`z` is the *host* matrix of eigenvectors,
`error` is the return code. `isAlreadyDecomposed` allows one can skip the decomposition if the `b` matrix stays the same between subsequent calls.

5.3.3 Auxillary routines

Important note for the GPU users. The overloaded convenience functions, like `cholesky()` can only be used if the data has been allocated on the host. If the data has been allocated on the GPU device, an automatic destinction of datatypes is not possible and one has to use the explicit functions specifying the datatype, e.g. `cholesky_double()`.

These auxillary routines are internally used by ELPA for transforming a generalized eigenvalue problem to a standard eigenvalue problem. Since these routines do offer GPU support (unlike in ScaLAPACK), and generally perform better also on CPUs than the respectice ScaLAPACK implementations, these routines are also available via the API. These procedures are:

`cholesky(a, error)`
: Overloaded function (for all datatypes) that returns the Cholesky decomposition for the *host* matrix `a`.

`cholesky_[double|float|double_complex|float_complex](a, error)`
: Explicit function (for all datatypes) that returns the Cholesky deomposition of the *host/device* matrix `a`.

`hermitian_multiply(uplo_a,uplo_c,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
: Overloaded function (for all datatypes) that multiplies the transposed/hermitian conjugated matrix A with matrix B and stores the result in matrix $C = A^{T/H}B$.
Here:
`uplo_a` is set to 'U' if A is upper triangular, 'L' if A is lower triangular, or anything else if A is a full matrix;
`uplo_c` is set to 'U' if C is upper triangular, 'L' if C is lower triangular, or anything else if C is a full matrix;
`ncb` is the number of columns of the global matrices `b` and `c`;
`a` is the *host* matrix A,
`b` is the *host* matrix B,
`nrows_b` is the number of rows of matrix `b`;
`ncols_b` is the number of columns of matrix `b`;
`c` is the *host* matrix C,
`nrows_c` is the number of rows of matrix `c`;
`ncols_c` is the number of columns of matrix `c`;
`error` is the return code.

`hermitian_multiply_[double|float|double_complex|float_complex]`
: `(uplo_a,uplo_c,ncb,a,b,nrows_b,ncols_b,c,nrows_c,ncols_c,error)`
Explicit function (for all datatypes) that multiplies the transposed/hermitian conjugated matrix `a` with matrix `b` and stores the results in matrix `c`. Arguments are the same as above except:
`a` is the *host/device* matrix A,
`b` is the *host/device* matrix B,
`c` is the *host/device* matrix C.

`invert_triangular(a, error_elpa)`

Overloaded function (for all datatypes) that inverts the upper triangular *host* matrix a.

`invert_triangular_[double|float|double_complex|float_complex](a, error_elpa)`
Explicit function (for all datatypes) that inverts the upper triangular *host/device* matrix a.

5.4 Using ELPA in a sequential code

For the simplest case of using ELPA in a sequential code, the following steps have to be taken:

1. Use the elpa module and create a pointer to an ELPA object

```
use elpa
class(elpa_t), pointer :: elpa
integer :: success
```

2. Initialize ELPA by passing the API version that is going to be used

```
if (elpa_init(20171201) /= ELPA_OK) then
  print *, "ELPA API version not supported"
  stop
endif
```

3. Allocate the ELPA object

```
elpa => elpa_allocate(success)
if (success /= ELPA_OK) then
  ! react on the error
  ! we urge every user to always check the error codes
  ! of all ELPA functions
endif
```

4. Specify the information about the input matrix via setting the *mandatory parameters*. Note that although a BLACS grid as such is not used for sequential execution, the `nblk` parameter must be set to a non-zero value.

```
! size of the na by na input matrix
call elpa%set("na", na, success)

! number of eigenvectors that should be computed, 1 <= nev <= na
call elpa%set("nev", nev, success)

! number of local rows of the matrix
call elpa%set("local_nrows", na, success)

! number of local columns of the matrix
call elpa%set("local_ncols", na, success)

! block size of the BLACS block-cyclic distribution
call elpa%set("nblk", nblk, success)
```

5. Call the `setup()` routine to complete the problem setup. This step finalizes the setting of *mandatory parameters* for the given ELPA object and they can not be changed in the future.

```
success = elpa%setup()
```

6. If desired, set any number of tunable *runtime options*. These can be changed between different calls of ELPA solver. A complete list of the runtime options can be found in Sec. 5.2.2.

```
call elpa%set("solver", ELPA_SOLVER_2STAGE, success)

! set the AVX BLOCK2 kernel; otherwise ELPA_2STAGE_REAL_DEFAULT will be
   used
call elpa%set("real_kernel", ELPA_2STAGE_REAL_AVX_BLOCK2, success)
```

7. Call the solver to obtain eigenvalues `ev(:)` and eigenvectors `z(:,:)`

```
call elpa%eigenvectors(a, ev, z, success)
```

8. Clean up by deallocating the ELPA object and uninitializing ELPA

```
call elpa_deallocate(elpa)
call elpa_uninit()
```

5.5 Using ELPA with OpenMP threads

If your hardware supports multi-threading, ELPA routines can take advantage of it. If also the math library supports multi-threading, it too can take advantage of the additional threads independently of ELPA. This is actually the case for Intel MKL, and there is in fact a sweet spot between the number of threads reserved for ELPA routines versus how many threads remain available to MKL. `TODO!!!` [Is the previous sentence correct?] The guideline for meeting the sweet spot is given later on in Sec. 6.2.

If ELPA has been built with OpenMP threading support, you can specify the number of OpenMP threads that ELPA will use internally. The steps involved in setting up the problem are the same as for sequential case (see Sec. 5.4) with one additional step: to allocate OpenMP threads for ELPA routines, it is **mandatory** to set the number of threads as a runtime parameter using the `set()` method in addition to setting it in the execution environment (via `export OMP_NUM_THREADS=...`):

```
call elpa%set("omp_threads", 4, error) ! set 4 threads for the elpa object
```

Note that to enable ELPA to use multi-threading, it should be configured with the switch `--enable-openmp=yes`. Needless to say, your compiler should of course support OpenMP.

For instructions on how to allocate hardware threads for the BLAS library, please refer to the reference guide of the library vendor. For Intel MKL, this is as easy as setting the environment variable `MKL_NUM_THREADS` to be equal to the desired number of threads. Please note that this number is just a suggestion to the library, and at runtime fewer threads may actually be used. For more details please see the Intel MKL developer guide.

5.6 Using ELPA with MPI

ELPA uses MPI to support the distributed-memory parallel execution model which also allows it to scale beyond one compute node. In this case, the distribution of the input matrix as well as the internal data follows the block-cyclic model provided by the BLACS library. Consequently, before calling ELPA, the user has to set up the BLACS grid and initialize the input matrix accordingly. The additional steps are summarized as follows.

1. Declare variables for the BLACS context and the ScaLAPACK descriptor

```
integer :: my_blacs_ctxt, sc_desc(problem_size)
```

2. MPI Initialization

```
call mpi_init(mpierr)
call mpi_comm_rank(mpi_comm_world,...)
call mpi_comm_size(mpi_comm_world,...)
```

3. Select the number of processor rows and columns. The application has to decide how the input matrix should be distributed. The grid setup may be done in an arbitrary way as long as it is consistent, i.e. $0 \leq$ `my_prow` $<$ `np_rows`, and $0 \leq$ `my_pcol` $<$ `np_cols`, and every process has a unique (`my_prow, my_pcol`) coordinate pair. For details see the documentation of `BLACS_Gridinit` and `BLACS_Gridinfo` of your BLACS installation. For better performance, it is recommended to setup the grid such that it is as close to a square grid as possible.

```
np_cols = some value
np_rows = some value
```

4. Set up the BLACS context and MPI communicators. The BLACS context is only necessary for using the ScaLAPACK routines (e.g. `numroc`, see below). For ELPA itself, the MPI communicators along rows and columns are sufficient.

```
my_blacs_ctxt = mpi_comm_world
call BLACS_Gridinit(my_blacs_ctxt, 'C', np_rows, np_cols )
call BLACS_Gridinfo(my_blacs_ctxt, np_rows, np_cols, my_prow, my_pcol)
```

5. For your distributed matrix, compute the number of local rows and columns per MPI task, e.g. with the ScaLAPACK routine `numroc`:

```
na_rows = numroc(na, nblk, my_prow, 0, np_rows)
na_cols = numroc(na, nblk, my_pcol, 0, np_cols)
```

6. Set up a BLACS descriptor for the target matrix

```
call descinit(sc_desc, na, na, nblk, nblk, 0, 0, my_blacs_ctxt, na_rows,
   info)
```

For ELPA the following restrictions hold:

- block sizes in both directions must be identical (arguments 4 and 5)
- first row and column of the distributed matrix must be on `p_row=0, p_col=0` (arguments 6 and 7)

Check also the return error code

```
if (info .ne. 0) then
  print *, "Invalid blacs-distribution. Abort!"
  stop
endif
```

7. Allocate space for the data and populate the input matrix for your problem to be solved

```
allocate( a(na_rows, na_cols) )
allocate( ev(na) )

a(i,j) = ...
```

8. Initialize ELPA and allocate ELPA object (similar to steps 2-3 in Sec. 5.4).

```
if (elpa_init(20211101) .ne. ELPA_OK) then
  print *, "ELPA API version not supported"
  stop
endif
elpa => elpa\_allocate(success)
if (success .ne. ELPA\_OK) then
  ! react on the error
  ! we urge users to always check the error codes of all ELPA functions
endif
```

9. Specify the information about the input matrix. Note that compared to step 4 in Sec. 5.4, there are three additional parameters that must be set, namely the MPI parent communicator, as well as the row and column indices for every processor:

```
! size of the matrix is na x na
call elpa%set("na", na, success)

! number of eigenvectors that should be computed, 1<= nev <= na
call elpa%set("nev", nev, success)

! number of local rows of the distributed matrix on this MPI task
call elpa%set("local_nrows", na_rows, success)

! number of local columns of the distributed matrix on this MPI task
call elpa%set("local_ncols", na_cols, success)

! block size for the BLACS block-cyclic distribution
call elpa%set("nblk", nblk, success)

! the global MPI communicator
call elpa%set("mpi_comm_parent", MPI_COMM_WORLD, success)

! row coordinate of MPI process
call elpa%set("process_row", my_prow, success)

! column coordinate of MPI process
call elpa%set("process_col", my_pcol, success)
```

10. From here on, the remaining steps are the same as steps 5 through 8 as outlined in the previous section. For the sake of clarity and to avoid confusion, we include them here as well:

```
success = elpa%setup()

! if desired, set any number of tunable run-time options
call elpa%set("solver", ELPA_SOLVER_2STAGE, success)
call elpa%set("real_kernel", ELPA_2STAGE_REAL_AVX_BLOCK2, success)

call elpa%eigenvectors(a, ev, z, success)

! cleanup
call elpa_deallocate(elpa)
call elpa_uninit()
```

For correctness, keep in mind to also call `mpi_finalize()` at the end of your program.

5.7 Hybrid MPI+OpenMP execution model

The steps needed to set up the program are the same as outlined previously in Sec. 5.6. Furthermore, it is important to note that in case of hybrid MPI and OpenMP execution, it is **mandatory** that your MPI library supports the threading levels `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`. You can check this e.g. by running one of the test programs, which will warn you if this prerequisite is not met. If your MPI library does not provide the required threading levels, ELPA will internally (independent of your applied setting) use only one OpenMP thread, and you will be informed at runtime with a warning. The number of threads used in a threaded implementation of your BLAS library will not be affected by this as long as these threads can be controlled through another method than specifying `OMP_NUM_THREADS` (for instance with Intel MKL library where you can specify `MKL_NUM\_THREADS`).

5.8 Using GPU acceleration

Currently, Nvidia, AMD, and Intel GPUs are supported. GPU acceleration is orthogonal to the previously mentioned execution models, meaning that each one can be further accelerated using GPUs. You have to make sure that ELPA has been configured with GPU support as explained earlier in Sec. 3.2. For sequential and multi-threaded runs, nothing more needs to be done. For MPI programs, more care has to be taken to ensure that the number of MPI tasks per GPU device is constant across all GPUs of a node. In case the total number of cores on a node is not divisible by the number of available GPUs (e.g. 20 cores and 3 GPUs), then one should leave some cores free, i.e. 18 active cores in the example above so that each GPU will serve 6 MPI tasks.

To maximize the achieved performance, it is recommended to have fewer MPI processes (ideally only one process) per GPU. However, this arrangement may harm the performance of the main application. One possible alternative is to use the extra available cores on each node as lightweight threads in the calling application (e.g. OpenMP, TBB, etc.). If this is not possible, then the application may use two different MPI communicators, one internal to the application itself which handles all the available physical cores, and one for ELPA containing as many MPI processes as there are GPU devices available. Consequently, the entire input matrix should be redistributed over these MPI processes that call ELPA. When redistributing the matrix, care has to be taken so that the first row and the first column of the redistributed matrix are located on the 0-th processors row and the 0-th processors column, respectively. Of course, it has to be tested whether the additional performance achieved is actually worth the extra effort that goes to communicator splitting and data redistribution. In any case, ELPA works correctly with any desired number of MPI processes per GPU device, and the discussion above is only for the sake of improving the performance.

It is, however, not uncommon to have multiple MPI processes per GPU device. In this case, with Nvidia GPUs, performance can be substantially improved if the Nvidia MPS server is activated on each node. However, the MPS daemon must be started exactly **once** per node. Some batch submission systems take care of this automatically. Check with your system administrator if this feature is provided; otherwise, the following mechanism can be used to set up MPS properly.

In the submission script, here using SLURM just as an example, we call the mpi launcher to run a **wrapper script**. This way, in the wrapper script, the process IDs can be queried where only one process (e.g. process 0) sets up the MPS server:

1. In the job submission script:

   ```
   # set up the environment
   ...
   srun ./wrapper_script.sh
   ```

2. In the `wrapper_script.sh`:

   ```
   #!/bin/bash
   # only process 0 sets up the MPS server:
   if [ $SLURM_LOCALID -eq 0 ]; then
     nvidia-cuda-mps-control -d
   fi

   # now launch the program
   ./<your_executable> <input_arguments>
   ```

More details on ELPA on GPUs and Nvidia MPS can be found here

5.9 Using ELPA from Python

In order to use ELPA from within your python code, a wrapper has to be generated, which allows you to import ELPA's functionality via a generated shared object. This requires that the two flags `--enable-python` and `--enable-python-tests` are specified at configure time. Then, after compilation using `make`, run the install command

```
make install DESTDIR=/absolute/path/to/install/dir
```

where `DESTDIR` should point to an absolute path where you have write permission. The install command will create a hierarchy of directories inside `DESTDIR`. The relative path we need for the next step is:

```
usr/local/lib/python3.x/site-packages
```

which must be included in your system's `PYTHONPATH` using e.g.

```
export PYTHONPATH=/DESTDIR/usr/local/lib/python3.x/site-packages:$PYTHONPATH
```

Note that 'x' refers to the minor version of your system's python3 installation, e.g. `python3.8` . Note also that, similar to using ELPA from a C or Fortran program, the path to where ELPA's shared libraries are generated must be known to the loader at runtime. This can be done using either the `rpath` mechanism, or by adding the path to the `LD_LIBRARY_PATH`. The exact path is: `/your/build/directory/.libs` .

Now you should be able to import the shared object into your python code:

```python
from pyelpa import DistributedMatrix
```

To actually use ELPA from python, there are a few steps to be taken to set up and solve the problem. The example code included in the repository under `elpa/python/examples/example.py` shows these steps. For the sake of brevity and to avoid repetition, it will not be included here. It is worthwhile to mention however, that as the example shows, there are two different ways to go over the elements of any matrix (in order to for example set up the input matrix). One is labelled to be the easiest yet less efficient where the elements are accessed individually one by one. The other method uses the block structure and is therefore more efficient. The difference in the efficiency of these methods would likely play a major role for accessing the elements of very large matrices.

6 Best practices

6.1 Autotuning for better performance

For autotuning, if the API version is set to a value below 20211125, the old autotuning implementation is used, and for the 20211125 version, the new implementation is used.

ELPA's autotuning engine is a powerful utility that can optimize a large number of tunable *runtime options*. Their optimal values can then be used for subsequent runs. This means that to obtain these values, ELPA needs to solve the problem once in order to test and compute the optimal tunable parameters. The first run will likely be sub-optimal however, and, therefore, autotuning is particularly promising if the problem has to be solved repeatedly as is the case of self-consistent methods for instance.

To use this feature, the application code must implement a few steps in similar way as explained earlier in Sec. 5. These steps are explained in the following paragraphs.

1. In your program decleration, declare the following two objects

   ```
   class(elpa_t), pointer          :: elpa
   class(elpa_autotune_t), pointer :: tune_state
   ```

2. Follow the steps needed to set up the problem as explained earlier in Sec. 5

3. After that, and before calling the solver, initialize the autotuning API version

   ```
   call elpa%autotune_set_api_version(20211125, success)
   assert_elpa_ok(success)
   ```

   The new version is 20211125. To use an older version, you can either set the API version to an older, supported one, or just not set it at all.

4. Initialize the tuning object

   ```
   tune_state => elpa%autotune_setup(tuning_level, tuning_domain, success)
   assert_elpa_ok(success)
   ```

   There are currently three possible levels for parameter tuning:

   - ELPA_AUTOTUNE_FAST, which includes the parameters related to the following items: solver, real_kernel, complex_kernel, omp_threads
   - ELPA_AUTOTUNE_MEDIUM, which, in addition to the above-mentioned parameters, includes the GPU-related ones
   - ELPA_AUTOTUNE_EXTENSIVE, includes all of the above parameters plus the ones related to the following items:
     various blocking factors, stripewidth_[real|complex], intermediate_bandwidth

   Furthermore, there are parameters that are relevant to real or complex problems only, while others are relevant to any problem type. The domain parameter controls whether tuning will be performed for real (ELPA_AUTOTUNE_DOMAIN_REAL) or complex (ELPA_AUTOTUNE_DOMAIN_COMPLEX) problems or for both cases through ELPA_AUTOTUNE_DOMAIN_ANY.

   The list of all tunable parameters can be obtained using a python script included in your current release. To do so, change directory to the following path

   ```
   cd elpa_dir/utils/parse_index
   ```

where `elpa_dir` is the main directory that includes all ELPA source files. Next, call the parser, which prints a list of parameter names and their description to standard output:

```
python extract_options.py
```

A complete list of these parameters for the current release is included in Sec. 5.2 and Appendix B along with a discussion on potential impacts of certain parameters on correctness and/or performance wherever necessary.

At this stage, if you wish to remove any of the tunble parameters from the tuning process[2], you should explicitly set the desired value before going to the next step. For example,

```
call elpa%set("solver", ELPA_SOLVER_2STAGE, success)
```

will remove the choice of solver from the set of tunable parameters.

5. Construct a loop in which the solver is iteratively called to solve the same problem until the tuning engine converges

   Remember to keep a copy of the input matrix which will have to be used to restore it because the solver overwrites the input.

```
do while (elpa%autotune_step(tune_state, success))
  assert_elpa_ok(success)

  call elpa%eigenvectors(a, ev, z, success)
  assert_elpa_ok(success)

  a(:,:) = a_copy(:,:)
end do
```

6. Set and print the optimal settings

   Once the tuning is done, the converged parameters can be set as the best combination by calling the subroutine `elpa%autotune_set_best()`. Afterwards, repeated calls to the solver will run using the optimal parameters.

```
call elpa%autotune_print_state(tune_state)
call elpa%autotune_save_state(tune_state, "saved_state.txt")

call elpa%autotune_set_best(tune_state, success)
assert_elpa_ok(success)

print *, "The best combination found by the autotuning:"
call elpa%autotune_print_best(tune_state, success)
assert_elpa_ok(success)
```

7. Finally, deallocate the objects and finalize the program

```
call elpa_autotune_deallocate(tune_state, success)
assert_elpa_ok(success)

call elpa_deallocate(elpa, success)
assert_elpa_ok(success)
```

[2]One reason might be that the user, for good reasons, wishes to fix the value of certain parameters, and, therefore, would like to speed up the tuning process by removing these parameters from the list of tunables.

```
call elpa_uninit(success)
assert_elpa_ok(success)
```

6.2 Using OpenMP threads

If ELPA is to be built with OpenMP support, irrespective of wheter MPI is enabled or not, please ensure that you also link it against a BLAS and LAPACK library which does offer threading support; otherwise, a severe performance loss will be encountered. Please refer to the documentation of your math library for details on multi-threading support and how to activate it.

To achieve an optimal distribution of OpenMP threads between ELPA and the math library, the following heuristic guideline can be used. Here, it is assumed that the matrix size is around 5k or above, and that the CPU consists of a total of 40 cores. For other configurations, the following numbers should be adjusted accordingly.

If Intel MKL is used, it is best to allow it to pick the number of threads dynamically by setting `MKL_DYNAMIC=TRUE`. Then, around 15-20 threads for ELPA2's internal solver should, to a good approximation, be in the optimal range. It can be set via the `OMP_NUM_THREADS` variable. For the ELPA1 solver, the above range should be reduced to 10-15 threads.

For other math libraries supporting multi-threading, a reasonable setup could be to allocate at least 10-12 threads for ELPA with at least around 8-10 threads left for the math library.

6.3 MPI

The following information holds for all runs of ELPA as long as MPI is used, including also pure-MPI and hybrid MPI-OpenMP runs.

For MPI runs, ELPA requires that matrices are distributed in a BLACS block-cyclic distribution. he BLACS matrix layout representation can be chosen to be either "row-major" or "column-major". The choice migth depend on the requirements of your application. ELPA works with both choices, but for the best performance, it might be necessary that both alternatives are tested. In case there is no special requirement from the application's perspecive, we recommend to use the "column-major" ordering.

Furthermore, the distribution of the MPI processes into a logical, 2D process grid should be specified. This setup is then used to address the BLACS block-cyclic distributed matrix with "row" and "column" processes. ELPA works correctly irrespective of the choice of the 2D processor grid, which is automatically deduced by ELPA from the underlying BLACS grid:

```
call BLACS_Gridinit(my_blacs_ctxt, layout, np_rows, np_cols)
```

However, the choice of the BLACS matrix layout (column- or row-major) and the 2D MPI processor grid (`np_rows, np_cols`) can affect the ELPA performance.

As an example: using 8 MPI processes, a 2D grid can be chosen to have `(np_rows=4, np_cols=2)`, or `(np_rows=2, np_cols=4)`. The best performance will be achieved using the following settings:

- `np_rows=4, np_cols=2` for column-major layout ("C")
- `np_rows=2, np_cols=4` for row-major layout ("R")

It's worth noting that ELPA comes with test programs located in the ELPA build folder. These programs are compiled when you run the `make` command during the ELPA installation. These tests can show how performance is affected if the BLACS block-cyclic distribution layout and the 2D processor grid aren't set optimally. For example, you can run:

```
mpiexec -n 8 ./validate_real_double_eigenvectors_2stage_random_all_layouts \
1000 1000 16
```

The test program calculates eigenvectors for real double-precision random matrix with its elements uniformly distrubuted on $[0, 1]$ interval, using ELPA2 solver and testing all BLACS layouts. Here the values "`1000 1000 16`" correspond to the matrix-size (`na`), the number of eigenvectors sought (`nev`), and the block size of BLACS block-cyclic distribution (`nblk`) respectively. Consequently, the timings for the solutions of the eigenvalue problem in all possible combinations of the layout and the 2D processor grid will be obtained.
CAUTION: test this only on small matrix sizes and a small number of MPI tasks; otherwise, the total runtime will be very large.

The ELPA test programs also provide detailed information about the settings as shown in the excerpt below:

```
Matrix size:  1000
Num eigenvectors:  1000
Blocksize:  16
Num MPI proc:  8
Number of processor rows=4, cols=2, total=8
Process layout:  C
```

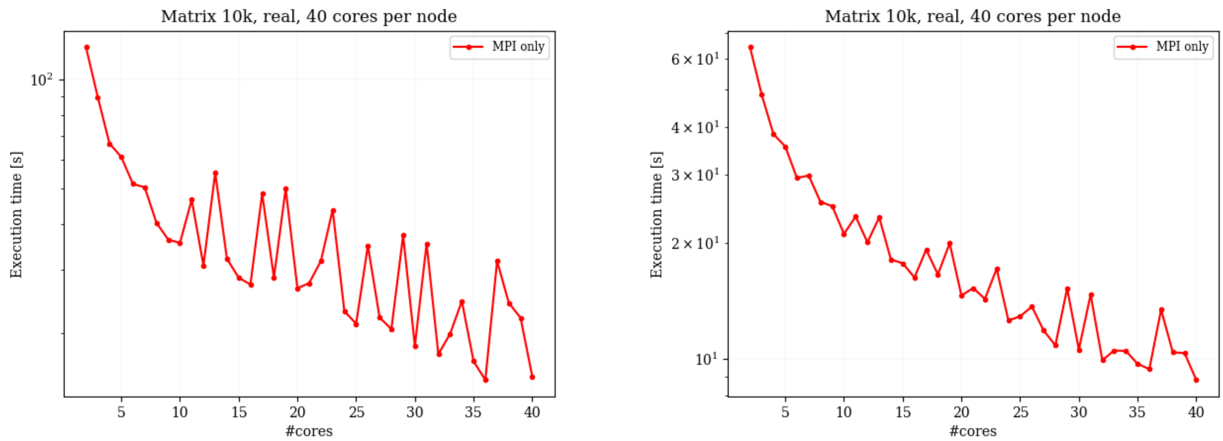
6.3.1 The 2D processor grid, quadratic setups and chosing the number of MPI tasks

As mentioned above, ELPA works correctly irrespective of the choice of the BLACS layout and the distribution of the 2D processor grid even if these are set in a sub-optimal way. However, some 2D processor grids lead to better performance than others. As a rule of thumb, ELPA solvers work best if the 2D processor grid (internal to ELPA) is quadratic or at least as "quadratic" as possible. For example, using 16 MPI tasks, the setup (MPI-rows=4, MPI-columns=4) works best. On the other hand, the following setups work correctly but with less-than-optimal performance:

- (8,2)
- (2,8)
- (16,1) $\rightarrow$ very bad
- (1,16) $\rightarrow$ very bad

Especially, very elongated setups with only one process row/column should be avoided. This also implies that the runtime of the solution can be influenced by the number of MPI tasks employed: in some situations it might be beneficial to use less MPI tasks than there are cores available in order to ensure that a well-shaped, (almost-)quadratic 2D grid can be set up. For example, on a hypothetical machine with 13 cores, one should not use all 13 MPI tasks as the only possible combination of `np_rows` and `np_cols` are 1 and 13. Rather, one should use 12 MPI tasks and leave one core idle to obtain a better distribution of 4×3.

The impact is illustrated in Figure 1 where the run-time for the solution of a real matrix (size 10k) with varying number of MPI processes from 2 to 40 is shown. For prime numbers, only very elongated process grids are possible, and a dramatic performance drop can be seen. Note that in all these tests, the choice of the number of processor rows and columns is always as optimal as possible. Please also note that this setup has been tuned to magnify the effect of the processor grid, and the execution times do not correspond to the optimal run-time as ELPA was built with no optimizations for this test.



(a) ELPA 1-stage (b) ELPA 2-stage

Figure 1: Performance impacts of the 2D processor layout

In case, when the external application has to run with a processor grid which is sub-optimal for ELPA, it might be beneficial to use the available funcionality of re-distributing the matrix and the processor grid (internal to ELPA) to obtain a better setup.

6.4 Hybrid MPI+OpenMP

For the optimal performance of ELPA in the hybrid MPI-OpenMP mode, it is important that the combination of the number of MPI tasks and OpenMP threads does not over-subscribe the compute nodes. Also, nested OpenMP regions should be disabled, and please also ensure that still more than one thread remains available for the math library (providing the BLAS and LAPACK functionality). Last but not least, please also make sure that the MPI tasks as well as the OpenMP threads per task are pinned in an appropriate way defined for your system. Consequently, the following requirements should be fulfilled:

1. The no. MPI tasks per node $\times$ the no. OpenMP threads per task must be $\leq$ no. cores per node
2. Set the number of OpenMP threads via the `OMP_NUM_THREADS` variable
3. Set the number of threads for the math library. For Intel MKL, `MKL_NUM_THREADS` should be set to a value larger than 1. Alternatively, `MKL_DYNAMIC=TRUE` should be used keeping in mind again, that more than one thread remains available for the math library
4. Process/thread migration should be prevented via correct pinning of MPI tasks and OpenMP threads, but do **not** pin to *hyperthreads*

If ELPA is to be built with Nvidia GPU support, please make sure that the configure command also explicitly sets the compute capability variable to the highest level supported by your hardware. For instance, for A100 devices, it should be set to `sm_80`. Please also ensure that, at the end of the configure step, the GPU kernels are listed.

Note that, depending on the matrix size and the number of MPI processes, the GPU version of both ELPA1 and ELPA2 solvers may compete closely in terms of performance. As a rule of thumb, ELPA1-GPU is ususally a better choice if 100% of eigenvectors are requested, while for small portion of eigenvectors, ELPA2-GPU should be preferred. Therefore, please test/autotune to find out which solver best suits the problem.

7 Troubleshooting

If you face any issues with using ELPA, the information in this section will help you find a solution.

7.1 Debugging information

It is very helpful to have debugging information for troubleshooting. To this end, please instruct ELPA to generate the extra details at run-time using the `set()` method after instantiating the elpa object as:

```
call elpa%set("debug", 1, error_elpa)
assert_elpa_ok(error_elpa)
```

Alternatively, if your code doesn't set the `debug` flag as described above, you can set the environment variable `export ELPA_DEFAULT_debug=1` either in your shell or in the slurm script before running the executable. In the event of an issue, please provide the developers with the reported debug information for troubleshooting. Please also follow the guidelines listed in Section 7.2.

7.2 Reporting bugs and issues

If you run into issues with using ELPA, you are welcome to contact us via elpa-library@mpcdf.mpg.de. However, please note that in order for us to to successfully find a solution as quick as possible, it is important that you provide the following information when you report an issue:

1. Information about the toolchain including which Fortran and C compiler and version as well as which math library were used. If applicable, also which MPI library and version, and which GPU compiler
2. The complete command that was used during the build process. Please **note** that it can be helpful to specify the configure flag `--enable-store-build-config` when configuring ELPA. It will compile the build configuration information into the library object, which can then be querried if needed
3. Information about the input data including matrix type and size
4. The error message and any extra debug information generated as explained in Sec. 7.1

8 Contributions guide

It has been and continues to be a tremendous effort to develop and maintain the ELPA library. Every help to improve ELPA is highly appreciated.

To open pull requests and issues, please use the ELPA repository on GitHub:
`https://github.com/marekandreas/elpa`
(which is a public mirror of ELPA's official repo `https://gitlab.mpcdf.mpg.de/elpa/elpa`)

For recommendations and suggestions, both for improving the code and the documentation, you can also send an e-mail to elpa-library@mpcdf.mpg.de.

Appendices

A Expert configure options

Here we list some additional, “expert” flags that can be specified during `configure` step. These flags are listed for completness; they are not needed in typical use cases.

`--enable-optional-argument-in-C-API`
: Make the error argument in the C API optional.
Default: `disabled`

`--enable-mpi-launcher`
: Specify an MPI launcher for executing small test programs during configuration and compilation. Must be combined with `--disable-detect-mpi-launcher`.
Example: `--enable-mpi-launcher=mpiexec`
Default: detect automatically (see `--disable-detect-mpi-launcher`)

`--disable-detect-mpi-launcher`
: Do not automatically detect the MPI launcher required to run small test programs during configuration and compilation.
Default: `enabled` (= detect launcher)

`--with-threading-support-check-during-build={yes|no}`
: Run a small program during configuration to check sufficient threading support of the MPI library. Disable only if launching this test program causes problems, for example, because you are not allowed to run an MPI program on the machine you are compiling ELPA on.
Default: `yes`

`--disable-runtime-threading-support-checks`
: **Use with caution!** Do not verify the required threading support (`MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`) of the MPI library at runtime. Disable only if you have verified the compatibility of the MPI library, otherwise ELPA will yield incorrect results without notification.
Default: `enabled`

`--disable-allow-thread-limiting`
: **Use with caution!** Do not reduce the number of OpenMP threads to 1 if the MPI library does not offer sufficient threading support (`MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`). Potentially causes incorrect results.
Default: `enabled`

`--disable-band-to-full-blocking`
: Use blocking implementation when transforming from band to full matrix.
Default: `enabled`

`--enable-autotune-redistribute-matrix`
: **Experimental!** During autotuning, re-distribute the matrix across the MPI ranks to find the optimal block size in the block-cyclic distribution. Requires the corresponding ScaLAPACK

functionality.
Default: `disabled`

B Expert key-value runtime option pairs for setting the ELPA object

Most commonly used rutime options are described in Sec. 5.2.2. Here we list additional runtime options that are considered to be expert settings. They are not needed in the typical use cases and documented here for completeness.

B.1 Expert general runtime options

The following are general runtime options.

`bandwidth`
: *Integer.* If set, the input matrix is assumed to be a band matrix with this bandwidth. Must be a multiple of `nblk`, but at least $2 \cdot$ `nblk`.

`cannon_for_generalized`
: `0` or `1`. Use Cannon's algorithm for the generalized eigenvalue problem.
Default: `1` (= enabled)

`cannon_buffer_size`
: *Integer.* If set, use this buffer size for Cannon's algorithm. Larger buffers accelerate the algorithm, but occupy more memory. Only relevant if `cannon_for_generalized` is `1`.

`qr`
: `0` or `1`. *For ELPA2.* Use QR decomposition. Only relevant for real matrices.
Default: `0` (= disabled)

`matrix_order`
: Either `COLUMN_MAJOR_ORDER` or `ROW_MAJOR_ORDER`. Define the matrix layout to be used when the matrix is re-distributed during autotuning. Only relevant if ELPA has been configured with `--enable-autotune-redistribute-matrix`. In all other cases the matrix layout is automatically deduced by ELPA from the underlying BLACS grid and this parameter is ignored.
Default: `COLUMN_MAJOR_ORDER`

`internal_nblk`
: *Integer.* Block size for the block-cyclic matrix layout used for re-distribution during autotuning. Only relevant if ELPA has been configured with `--enable-autotune-redistribute-matrix`.

`check_pd`
: `0` or `1`. Check if all computed eigenvalues are greater than `thres_pd_double` or `thres_pd_single`, depending on the precision used.
Default: `0` (= disabled)

`thres_pd_double`
: *Float.* Lower bound of eigenvalues computed in double precision. Only relevant if `check_pd` is `1`.
Default: 10^{-5}

`thres_pd_single`
: *Float.* Lower bound of eigenvalues computed in single precision. Only relevant if `check_pd` is `1`.
Default: 10^{-5}

`intermediate_bandwidth`
: *Integer. For ELPA2.* Intermediate bandwidth used for conversion to band matrix form.

	Default: $\max\{64, \texttt{nblk}\}$ for real matrices, $\max\{32, \texttt{nblk}\}$ for complex matrices
`min_tile_size`	*Integer.* Minimum tile size used in tridiagonalization (ELPA1) and band reduction (ELPA2). **Default:** $128 \cdot \max\{\texttt{np\_rows}, \texttt{np\_cols}\}$
`blocking_in_band_to_full`	*Integer.* Blocking factor when transforming from band to full matrix. Only relevant if ELPA has been configured with `--enable-band-to-full-blocking`. **Default:** `3`
`max_stored_rows`	*Integer.* Maximum number of rows stored in ELPA1 backtransformation. **Default:** `256`
`stripewidth_real`	*Integer.* `TODO!!!` Must be a multiple of 4. **Default:** `48`
`stripewidth_complex`	*Integer.* `TODO!!!` Must be a multiple of 8. **Default:** `96`

B.2 Expert runtime options for GPU

The following are expert runtime options related to GPUs. All flags can be enabled or disabled by setting them to `1` or `0`, respectively.

`gpu`	**Deprecated.** Enable GPU acceleration using Nvidia GPUs. Please use explicit parameters for the various vendors instead, e.g. 'nvidia-gpu', 'amd-gpu', or 'intel-gpu', since this option is depricated and will be disabled in the future. **Default:** `0` (= disabled)
`gpu_hermitian_multiply`	Compute matrix-matrix multiplications on GPUs. **Default:** `1` (= enabled)
`gpu_invert_trm`	Compute inversion of upper triangular matrices on GPUs. **Default:** `1` (= enabled)
`gpu_cholesky`	Compute Cholesky factorization on GPUs. **Default:** `1` (= enabled)
`gpu_tridiag`	Tridiagonalize matrix using GPUs. **Default:** `1` (= enabled)
`gpu_solve_tridi`	Solve the eigenproblem for tridiagonal matrix on GPUs. **Default:** `1` (= enabled)
`gpu_trans_ev`	*For ELPA1.* Compute eigenvector transformation from tridiagonal to full matrix representation on GPUs. **Default:** `1` (= enabled)
`gpu_bandred`	*For ELPA2.* Compute reduction to band matrix on GPUs. **Default:** `1` (= enabled)

gpu_trans_ev_tridi_to_band
: *For ELPA2.* Compute eigenvector transformation from tridiagonal to band matrix representation on GPUs.
Default: 1 (= enabled)

gpu_trans_ev_band_to_full
: *For ELPA2.* Compute eigenvector transformation from band to full matrix representation on GPUs.
Default: 1 (= enabled)

B.3 Expert runtime options for collective MPI operations

The runtime options in this section control the communication pattern in ELPA. They allow switching from blocking to non-blocking communication (NBC) for collective operations for certain parts of the library. All flags are disabled by default and can be enabled by setting them to 1.

nbc_row_global_gather
: Use NBC for rows in global_gather.

nbc_col_global_gather
: Use NBC for columns in global_gather.

nbc_row_global_product
: Use NBC for rows in global_product.

nbc_col_global_product
: Use NBC for columns in global_product.

nbc_row_solve_tridi
: Use NBC for rows in solve_tridi.

nbc_row_transpose_vectors
: Use NBC for rows in transpose_vectors.

nbc_col_transpose_vectors
: Use NBC for columns in transpose_vectors.

nbc_row_herm_allreduce
: Use NBC for rows in herm_allreduce.

nbc_col_herm_allreduce
: Use NBC for columns in herm_allreduce.

nbc_row_sym_allreduce
: Use NBC for rows in sym_allreduce.

nbc_col_sym_allreduce
: Use NBC for columns in sym_allreduce.

nbc_row_elpa1_full_to_tridi
: *For ELPA1.* Use NBC for rows in tridiag.

nbc_col_elpa1_full_to_tridi
: *For ELPA1.* Use NBC for columns in tridiag.

nbc_row_elpa1_tridi_to_full
: *For ELPA1.* Use NBC for rows in trans_ev.

`nbc_col_elpa1_tridi_to_full`
: *For ELPA1.* Use NBC for columns in `trans_ev`.

`nbc_row_elpa2_full_to_band`
: *For ELPA2.* Use NBC for rows in `bandred`.

`nbc_col_elpa2_full_to_band`
: *For ELPA2.* Use NBC for columns in `bandred`.

`nbc_all_elpa2_band_to_tridi`
: *For ELPA2.* Use NBC in `tridiag_band`.

`nbc_row_elpa2_tridi_to_band`
: *For ELPA2.* Use NBC for rows in `trans_ev_tridi_to_band`.

`nbc_col_elpa2_tridi_to_band`
: *For ELPA2.* Use NBC for columns in `trans_ev_tridi_to_band`.

`nbc_row_elpa2_band_to_full`
: *For ELPA2.* Use NBC for rows in `trans_ev_band_to_full`.

`nbc_col_elpa2_band_to_full`
: *For ELPA2.* Use NBC for columns in `trans_ev_band_to_full`.

`nbc_all_elpa2_redist_band`
: *For ELPA2.* Use NBC in `redist_band`.

`nbc_all_elpa2_main`
: *For ELPA2.* Use NBC in `elpa_solve_ev`.

C ELPA functions

In this Appendix, we list all ELPA math and auxillary functions and their arguments. This Appendix is a copy of the *man* pages provided with every ELPA installation. They can be invoked by a shell command from the `./man` folder that is located in the elpa root directory, for example:

```
git clone https://gitlab.mpcdf.mpg.de/elpa/elpa.git
cd elpa/man
ls # list all available man pages
man ./elpa_eigenvalues.3
```

for showing the man page for `eigenvalues()` routine.

C.1 elpa2_print_kernels

NAME

elpa2_print_kernels − provides information, which ELPA2 kernels are available on this system.

SYNOPSIS

elpa2_print_kernels

Description

Provides information, which ELPA2 kernels are available on this system.
It is possible to configure ELPA2 such, that different compute intensive ’ELPA2 kernels’ can be chosen at runtime. The service binary elpa2_print_kernels will query the library and tell whether ELPA2 has been configured in this way, and if this is the case which kernels can be chosen at runtime. It will furthermore detail whether ELPA has been configured with OpenMP support.

Options

none

Author

A. Marek, MPCDF

Reporting bugs

Report bugs to the ELPA mail elpa-library@mpcdf.mpg.de

SEE ALSO

elpa_init(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.2 elpa_allocate

NAME

elpa_allocate − allocates an instance of the ELPA library

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

elpa => **elpa_allocate** (error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
Returns an instance of the ELPA object

integer, optional :: **error**
A returned error code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

elpa_t handle = **elpa_allocate**(**int** *error);

With the definitions of the input and output variables:

elpa_t **handle**; // returns an handle to the allocated ELPA object

int ***error**; // a returned error code

DESCRIPTION

Allocate an ELPA object. The function **elpa_init**(3) must be called once *BEFORE* **elpa_allocate** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.3 elpa_autotune_deallocate

NAME

elpa_autotune_deallocate − deallocates an ELPA autotuning instance

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa class(elpa_autotune_t), pointer :: tune_state

call elpa%**autotune_deallocate** (tune_state, error)

With the definitions of the input and output variables:

type(elpa_autotune_t) :: **tune_state**
The ELPA autotuning object, created with **elpa_autotune_setup**(3)

integer, optional :: **error**
The returned error code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

void **elpa_autotune_deallocate** (**elpa_t** handle, **elpa_autotune_t** autotune_handle, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

elpa_autotune_t **autotune_handle**;
The handle of an ELPA object, obtained before with **elpa_autotune_setup**(3)

int ***error**;
The returned error code

DESCRIPTION

Deallocates an ELPA autotuning instance. *Prior* to calling the elpa_autotune_deallocate method, an ELPA autotuning object must have been created. See **elpa_autotune_setup**(3)

SEE ALSO

elpa_autotune_step(3) **elpa_autotune_setup**(3) **elpa_autotune_deallocate(3)**

C.4 elpa_autotune_load_state

NAME

elpa_autotune_load_state − loads a state of an ELPA autotuning object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa class(elpa_autotune_t), pointer :: autotune

call elpa%**autotune_load_state** (autotune, filename, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object

class(elpa_autotune_t) :: **autotune**
An instance of the ELPA autotune object

character(*) ::**filename**
The filename to be used for loading the settings

integer, optional :: **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

void **elpa_autotune_load_state**(**elpa_t** handle, **elpa_autotune_t** autotune_handle, **const char** *filename, **int** *error):

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

elpa_autotune_t **handle**;
The handle to the ELPA autotune object

const char ***filename**;
The filename to load the settings

int ***error**;
The error return code

DESCRIPTION

Loads a previously stored state of an autotune object. With the loaded, state the autotuning could be resumed.

SEE ALSO

elpa_autotune_save_state(3)

C.5 elpa_autotune_print_state

NAME

elpa_autotune_print_state − prints the current state of an ELPA autotuning object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa
class(elpa_autotune_t), pointer :: autotune

call elpa%**autotune_print_state** (autotune, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object

class(elpa_autotune_t) :: **autotune**
An instance of the ELPA autotune object

integer, optional :: **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

void **elpa_autotune_print_state**(**elpa_t** handle, **elpa_autotune_t** autotune_handle, **int** *error):

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

elpa_autotune_t **handle**;
The handle to the ELPA autotune object

int ***error**;
The error return code

DESCRIPTION

Prints the current state of an autotune object.

SEE ALSO

elpa_autotune_save_state(3) **elpa_autotune_load_state**(3)

C.6 elpa_autotune_save_state

NAME

elpa_autotune_save_state − saves the current state of an ELPA autotuning object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa class(elpa_autotune_t), pointer :: autotune

call elpa%**autotune_save_state** (autotune, filename, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object

class(elpa_autotune_t) :: **autotune**
An instance of the ELPA autotune object

character(*) ::**filename**
The filename to be used for storing the settings

integer, optional :: **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

void **elpa_autotune_save_state**(**elpa_t** handle, **elpa_autotune_t** autotune_handle, **const char** *filename, **int** *error):

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

elpa_autotune_t **handle**;
The handle to the ELPA autotune object

const char ***filename**;
The filename to store the settings

int ***error**;
The error return code

DESCRIPTION

Saves the current state of an autotune object. The state can be restored with **elpa_autotune_load_state**(3) and the autotuning could be resumed.

SEE ALSO

elpa_autotune_load_state(3)

C.7 elpa_autotune_set_best

NAME

elpa_autotune_set_best − sets the tunable parameters to the up-to-now best solution
Before the autotuning options can be set, an autotuning step has to be done **elpa_autotune_step**(3)

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa
class(elpa_autotune_t), pointer :: tune_state

call elpa%**autotune_set_best** (tune_state)

With the definitions of the input and output variables:

type(elpa_autotune_t) :: **tune_state**
The ELPA autotuning object, created with **elpa_autotune_setup**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

void **elpa_autotune_set_best** (**elpa_t** handle, **elpa_autotune_t** autotune_handle);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

elpa_autotune_t **autotune_handle**;
The handle of an ELPA object, obtained before with **elpa_autotune_setup**(3)

DESCRIPTION

Sets the up-to-now best options for ELPA tunable parameters. *Prior* to calling the elpa_autotune_set_best method, an ELPA autotuning step must have been performed. See **elpa_autotune_set_best**(3)

SEE ALSO

elpa_autotune_step(3) **elpa_autotune_setup**(3) **elpa_autotune_deallocate(3)**

C.8 elpa_autotune_setup

NAME

elpa_autotune_setup − creates an instance for autotuning of the ELPA library

Before the autotuning object can be created, an instance of the ELPA library has to be setup, see e.g. **elpa_setup**(3)

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa class(elpa_autotune_t), pointer :: tune_state

tune_state= elpa%**autotune_setup** (level, domain)

With the definitions of the input and output variables:

integer :: **level**
The level of the autotuning, at the moment ELPA_AUTOTUNE_FAST is supported

integer :: **domain**
The domain (real or complex) of the autotuning, can be either ELPA_AUTOTUNE_DOMAIN_REAL or ELPA_AUTOTUNE_DOMAIN_COMPLEX

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

elpa_autotune_t autotune_handle = **elpa_autotune_setup** (**elpa_t** handle, int level, int domain);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

int **level**;
The level of the autotuning, at the moment "ELPA_AUTOTUNE_FAST" is supported

int **domain**;
The domain (real or complex) of the autotuning, can be either "ELPA_AUTOTUNE_DOMAIN_REAL" and "ELPA_AUTOTUNE_DOMAIN_COMPLEX

elpa_autotune_t **autotune_handle**;
The created handle of the autotune object

DESCRIPTION

Creates an ELPA autotuning object. *Prior* to calling the autotune_setup, an ELPA object must have been created. See**elpa_setup**(3)

SEE ALSO

elpa_autotune_step(3) **elpa_autotune_set_best**(3) **elpa_autotune_deallocate(3)**

C.9 elpa_autotune_step

NAME

elpa_autotune_step − does one ELPA autotuning step
Before the autotuning step can be done, an instance of the ELPA autotune object has to be created, see **elpa_autotune_setup**(3)

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa
class(elpa_autotune_t), pointer :: tune_state

unfinished = elpa%**autotune_step** (tune_state)

With the definitions of the input and output variables:

type(elpa_autotune_t) :: **tune_state**
The ELPA autotuning object, created with **elpa_autotune_setup**(3)

logical ::**unfinished**
Logical, specifying whether autotuning has finished (.false.) or not (.true.)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;
elpa_autotune_t autotune_handle;

int unfinished = **elpa_autotune_step** (**elpa_t** handle, **elpa_autotune_t** autotune_handle);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

elpa_autotune_t **autotune_handle**;
The handle of the autotuning object, created with **elpa_autotune_setup**(3)

int **unfinished**;
Integer, specifying whether autotuning has finished (0) or not (1)

DESCRIPTION

Performs an ELPA autotuning step. *Prior* to calling the autotune_step, an ELPA autotune object must have been created. See **elpa_autotune_setup**(3)

SEE ALSO

elpa_autotune_setup(3) **elpa_autotune_set_best**(3) **elpa_autotune_deallocate(3)**

C.10 elpa_cholesky

NAME

elpa_cholesky − does a Cholesky factorization of a real symmetric or complex hermitian matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_cholesky_double, elpa_cholesky_float, elpa_cholesky_double_complex, elpa_cholesky_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**cholesky** (a, error)

With the definitions of the input and output variables:

datatype :: **a** ! can also be a device pointer of type(c_ptr)
: The host/device matrix **a** which should be decomposed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

integer, optional :: **error**
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_cholesky**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

datatype ***a**; // can also be a device pointer
: The host/device matrix **a** which should be decomposed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex".

int ***error**;
: The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the Cholesky decomposition of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_cholesky** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal(3) elpa_hermitian_multiply(3) elpa_uninit(3) elpa_deallocate(3)**

C.11 elpa_deallocate

NAME

elpa_deallocate − deallocates an instance of the ELPA library after usage

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call **elpa_deallocate** (elpa, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
The pointer to the instance of the ELPA library that is to be deallocated

integer, optional :: **error**
The returned error code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void elpa_deallocate(**elpa_t** handle, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA instance which should be deallocated.

int ***error**;
The returned error code

DESCRIPTION

Deallocate an ELPA object. The functions **elpa_init**(3) and **elpa_allocate**(3) must have been called *BEFORE* **elpa_deallocate** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3)

C.12 elpa_eigenvalues

NAME

elpa_eigenvalues − computes all eigenvalues of a real symmetric or complex hermitian matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_eigenvalues_double, elpa_eigenvalues_float, elpa_eigenvalues_double_complex, elpa_eigenvalues_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvalues** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a**
The matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)". The matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real-valued eigenvalues.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex". The matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float". Note that the eigenvalues of complex hermitian

matrices are also real.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.13 elpa_eigenvalues_double

NAME

elpa_eigenvalues_double − computes all eigenvalues of a real double-precision symmetric matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvalues_double** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_double)". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to a matrix **a** in the device memory.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** must be "real(kind=c_double)". In case of a GPU build **ev** can be a device pointer of type "type(c_ptr)" to the vector of eigenvalues in the device memory

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues_double**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** must be "double". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** must be "double". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues of a double precision real symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues_double** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.14 `elpa_eigenvalues_double_complex`

NAME

elpa_eigenvalues_double_complex − computes all eigenvalues of a complex double-precision hermitian matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvalues_double_complex** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_double_complex)". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

datatype :: **ev**
The host/device vector of eigenvalues **ev** stored in *ascending* order. The number of requested eigenvalues must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the vector **ev** must be "real(kind=c_double)". In case of a GPU build **ev** can be a device pointer of type "type(c_ptr)" to the vector of eigenvalues in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues_double_complex**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double complex". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to matrix **a** in the device memory.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** must be "double". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

int ***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues of a double precision complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues_double_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.15 elpa_eigenvalues_float

NAME

elpa_eigenvalues_float − computes all eigenvalues of a real single-precision symmetric matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvalues_float** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_float)". The matrix has to be symmetric this is not checked by the routine. In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to a matrix **a** in the device memory.

datatype :: **ev**
The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the vector **ev** must be "real(kind=c_float)". In case of a GPU build **ev** can be a device pointer of type "type(c_ptr)" to the vector of eigenvalues in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues_float**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** must be "float". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;
The host/device storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** must be "float". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues of a single-precision real symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues_float** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.16 elpa_eigenvalues_float_complex

NAME

elpa_eigenvalues_float_complex − computes all eigenvalues of a complex hermitian single-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvalues_float_complex** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_float_complex)". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** must be "real(kind=c_float)". In case of a GPU build **ev** can be a device pointer of type "type(c_ptr)" to the vector of eigenvalues in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues_float_complex**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float complex". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to matrix **a** in the device memory.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** must be "float". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

int ***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues of a single-precision complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvalues_float_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.17 elpa_eigenvectors

NAME

elpa_eigenvectors − computes the eigenvalues and (part of) the eigenvector spectrum for a real symmetric or complex hermitian matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_eigenvectors_double, elpa_eigenvectors_float, elpa_eigenvectors_double_complex, elpa_eigenvectors_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvectors** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a**
The matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)". The matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype :: **ev**
The vector of eigenvalues **ev** stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real-valued eigenvalues.

datatype :: **q**
The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;

The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex". The matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype ***ev**;

The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float". Note that the eigenvalues of complex hermitian matrices are also real.

datatype ***q**;

The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex".

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvectors** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.18 elpa_eigenvectors_double

NAME

elpa_eigenvectors_double − computes all eigenvalues and (part of) the eigenvector spectrum for a real symmetric matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvectors_double** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which all eigenvalues and (part of) eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_double)". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype :: **ev**
The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the vector **ev** must be "real(kind=c_double)". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype :: **q**
The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix must be "real(kind=c_double)". In case of a GPU build **q** can be a device pointer to the matrix **q** in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvectors_double**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The host/device matrix **a** for which the eigenpairs should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;

The host/device storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The**datatype** must be "double". In case of a GPU b uild **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype ***q**;

The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** must be one of "double". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a real symmetric double precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvectors_double** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.19 elpa_eigenvectors_double_complex

NAME

elpa_eigenvectors_double_complex − computes all eigenvalues and (part of) the eigenvector spectrum for a complex hermitian matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvectors_double_complex** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues and eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix must be "complex(kind=c_double_complex)". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype :: **ev**
The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the vector **ev** must be "real(kind=c_double)". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype :: **q**
The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix must be "complex(kind=c_double_complex)". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvectors_double_complex**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues and eigenvectors should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double complex". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;

The host/device storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The**datatype** must be "double". In case of a GPU b uild **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype ***q**;

The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** must be one of "double complex". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a complex hermitian double precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvectors_double_complex** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.20 elpa_eigenvectors_float

NAME

elpa_eigenvectors_float − computes all eigenvalues and (part of) the eigenvector spectrum for a real symmetric single-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvectors_float** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues and (part of) eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_float)". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** must be "real(kind=c_float)". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype :: **q**
The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_float)". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvectors_float**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float". The matrix has to be symmetric, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;

The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** must be "float". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype ***q**;

The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be one of "float". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a real symmetric single-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvectors_float** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.21 elpa_eigenvectors_float_complex

NAME

elpa_eigenvectors_float_complex − computes all eigenvalues and (part of) the eigenvector spectrum for a complex hermitian single-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**eigenvectors_float_complex** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a** ! can also be a device pointer of type(c_ptr)
The host/device matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_float_complex)". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype :: **ev**
The host/device vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the vector **ev** must be "real(kind=c_float)". In case of a GPU build **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype :: **q**
The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** of the matrix must be "complex(kind=c_float_complex)". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvectors_float_complex**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float complex". The matrix has to be hermitian, this is not checked by the routine. In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***ev**;

The host/device storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The**datatype** must be "float". In case of a GPU b uild **ev** can be a device pointer to the vectors of eigenvalues in the device memory.

datatype ***q**;

The host/device storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** must be one of "float complex". In case of a GPU build **q** can be a device pointer to a matrix **q** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a complex hermitian single-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_eigenvectors_float_complex** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_skew_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.22 elpa_generalized_eigenvalues

NAME

elpa_generalized_eigenvalues − computes all eigenvalues of a generalized eigenvalue problem, A*X=lambda*B*X, for real symmetric or complex hermitian matrices

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**generalized_eigenvalues** (a, b, ev, is_already_decomposed, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a**
The matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

datatype :: **b**
The matrix **b** defining the generalized eigenvalue problem. The dimensions and datatype of the matrix **b** has to be the same as for matrix **a**.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real-valued eigenvalues.

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_generalized_eigenvalues**(**elpa_t** handle, **datatype** *a, **datatype** *b, **datatype** *ev, **int** is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of "double", "float", "double complex", or "float complex".

datatype * **b**;
  The matrix **b** defining the generalized eigenvalue problem. The dimensions and the **datatype** of the matrix **b** must be the same as matrix a.

datatype ***ev**;
  The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float". Note that the eigenvalues of complex hermitian matrices are also real.

int **is_already_decomposed**;
  Has to be set to 0 for the first call with a given **b** and 1 for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

int ***error**;
  The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the generalized eigenvalues and (part of) the eigenvector spectrum of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_generalized_eigenvalues** can be called. In particular, the number of eigenvectors to be computed, "nev", must be set with **elpa_set**(3). Unlike in the case of ordinary eigenvalue problem, the generalized problem calls some external ScaLAPACK routines. The user is responsible for initialization of the BLACS context, which then has to be passed to elpa by **elpa_set**(3) *BEFORE* **elpa_generalized_eigenvalues** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.23 elpa_generalized_eigenvectors

NAME

elpa_generalized_eigenvectors − computes all eigenvalues and (part of) eigenvectors of a generalized eigenvalue problem, A*X=lambda*B*X, for real symmetric or complex hermitian matrices

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**generalized_eigenvectors** (a, b, ev, q, is_already_decomposed, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a**
The matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)

datatype :: **b**
The matrix **b** defining the generalized eigenvalue problem. The dimensions and datatype of the matrix **b** has to be the same as for matrix a.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix. Note that complex hermitian matrices also have real-valued eigenvalues.

datatype :: **q**
The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

logical :: **is_already_decomposed**
Has to be set to .false. for the first call with a given **b** and .true. for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_generalized_eigenvectors**(**elpa_t** handle, **datatype** *a, **datatype** *b, **datatype** *ev, **datatype** *q, **int** is_already_decomposed, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which all eigenvalues and (part of) eigenvectors should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The **datatype** can be one of "double", "float", "double complex", or "float complex".

datatype * **b**;
The matrix **b** defining the generalized eigenvalue problem. The dimensions and the **datatype** of the matrix **b** must be the same as matrix **a**.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float". Note that the eigenvalues of complex hermitian matrices are also real.

datatype ***q**;
The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex".

int **is_already_decomposed**;
Has to be set to 0 for the first call with a given **b** and 1 for each subsequent call with the same **b**, since **b** then already contains decomposition and thus the decomposing step is skipped.

int ***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the generalized eigenvalues and (part of) the eigenvector spectrum of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_generalized_eigenvalues** can be called. In particular, the number of eigenvectors to be computed can be set with **elpa_set**(3). Unlike in the case of ordinary eigenvalue problem, the generalized problem calls some external ScaLAPACK routines. The user is responsible for initialization of the BLACS context, which then has to be passed to ELPA by **elpa_set**(3) *BEFORE* **elpa_generalized_eigenvalues** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.24 elpa_get_communicators

NAME

elpa_get_communicators − splits the global MPI communicator mpi_comm_global communicator into rows and column communicators mpi_comm_rows and mpi_comm_cols

SYNOPSIS

FORTRAN INTERFACE

use elpa1

success = **elpa_get_communicators** (mpi_comm_global, my_prow, my_pcol, mpi_comm_rows, mpi_comm_cols)

integer, intent(in) :: **mpi_comm_global**
Global communicator for the calculation

integer, intent(in) :: **my_prow**
Row coordinate of the calling process in the process grid

integer, intent(in) :: **my_pcol**
Column coordinate of the calling process in the process grid

integer, intent(out) :: **mpi_comm_rows**
Communicator for communication within rows of processes

integer, intent(out) :: **mpi_comm_cols**
Communicator for communication within columns of processes

integer ::**success**
Return value indicating success or failure of the underlying MPI_COMM_SPLIT function

C INTERFACE

#include "elpa_generated.h

success = **elpa_get_communicators** (int mpi_comm_world, int my_prow, int my_pcol, int *mpi_comm_rows, int *mpi_comm_cols);

int **mpi_comm_global**;
Global communicator for the calculation

int **my_prow**;
Row coordinate of the calling process in the process grid

int **my_pcol**;
Column coordinate of the calling process in the process grid

int ***mpi_comm_rows**;
Pointer to the communicator for communication within rows of processes

int ***mpi_comm_cols**;
Pointer to the communicator for communication within columns of processes

int **success**;
Return value indicating success or failure of the underlying MPI_COMM_SPLIT function

DESCRIPTION

All ELPA routines need MPI communicators for communicating within rows or columns of processes. These communicators are created from the **mpi_comm_global** communicator. It is assumed that the matrix used in ELPA is distributed with **my_prow** rows and **my_pcol** columns on the calling process. This function has to be invoked by all involved processes before any other calls to ELPA routines.

SEE ALSO

elpa_get_communicators(3) **elpa_solve_evp_real**(3) **elpa_solve_evp_complex**(3) **elpa2_print_kernels**(1)

C.25 elpa_hermitian_multiply

NAME

elpa_hermitian_multiply − performs a "hermitian" multiplication of matrices: C = A**T * B for real matrices and C = A**H * B for complex matrices

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_hermitian_multiply_triangular_double, elpa_hermitian_multiply_triangular_float, elpa_hermitian_multiply_triangular_double_complex, elpa_hermitian_multiply_triangular_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**hermitian_multiply** (uplo_a, uplo_c, ncb, a, b, nrows_b, ncols_b, &
c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

character*1 ::**uplo_a**
Should be set to ’U’ if A is upper triangular, to ’L’ if A is lower triangular or to anything else if A is a full matrix.

character*1 ::**uplo_c**
Should be set to ’U’ if only the upper diagonal part of C is needed, to ’L’ if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

integer ::**ncb**
The number of columns of the global matrices **b** and **c**.

datatype ::**a**
The matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

datatype ::**b**
The matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer ::**nr ows_b**
The number of rows of matrix **b**.

integer ::**ncols_b**
The number of columns of matrix **b**.

datatype ::**c**
The matrix **c**. The dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. The **datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer ::**nr ows_c**
The number of rows of matrix **c**.

integer ::**ncols_c**
The number of columns of matrix **c**.

integer, optional :: **error**

The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_hermitian_multiply**(**elpa_t** handle, **char** uplo_a, **char** uplo_c, **int** ncb, **datatype** *a, **datatype** *b, **int** nrows_b, **int** ncols_b, **datatype** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;

The handle to the ELPA object

char **uplo_a**;

Should be set to ’U’ if A is upper triangular, to ’L’ if A is lower triangular or anything else if A is a full matrix.

char **uplo_c**;

Should be set to ’U’ if only the upper diagonal part of C is needed, to ’L’ if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

int **ncb**;

The number of columns of the global matrices **b** and **c**.

datatype ***a**;

The matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "double", "float", "double comple x", or "float complex".

datatype ***b**;

The matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The **datatype** of the matrix can be one of "double", "float", "double complex", or "float complex".

int **nrows_b**;

The number of rows of matrix **b**.

int **ncols_b**;

The number of columns of matrix **b**.

datatype ***c**;

The matrix **c**. The dimensions of the matrix are specified by the parameters**n rows_c** and **ncols_c**. The **datatype** of the matrix can be one of "double", "float", "double complex", or "float complex".

int **nrows_c**;

The number of rows of matrix **c**.

int **ncols_c**;

The number of columns of matrix **c**.

int ***error**;

The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a "hermitian" multiplication: C = A**T * B for real matrices and C=A**H * B for complex matrices. The functions**elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.26 elpa_hermitian_multiply_double

NAME

elpa_hermitian_multiply_double − performs a "hermitian" multiplication of real double-precision matrices: C = A**T * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**hermitian_multiply_double** (uplo_a, uplo_c, ncb, a, b, nrows_b, ncols_b, &
c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

character*1 ::**uplo_a**
Should be set to ’U’ if A is upper triangular, to ’L’ if A is lower triangular or to anything else if A is a full matrix.

character*1 ::**uplo_c**
Should be set to ’U’ if only the upper diagonal part of C is needed, to ’L’ if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

integer ::**ncb**
The number of columns of the global matrices **b** and **c**.

datatype ::**a**
The host/device matrix **a**. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_double)". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ::**b**
The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** of the matrix must be "real(kind=c_double)". In case of a GPU b uild **b** can be a device pointer to a matrix **b** in the device memory.

integer ::**nr ows_b**
The number of rows of matrix **b**.

integer ::**ncols_b**
The number of columns of matrix **b**.

datatype ::**c**
The host/device matrix **c**. The dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. The**datatype** of the matrix must be "real(kind=c_double)". In case of a GPU b uild **c** can be a device pointer to a matrix **c** in the device memory.

integer ::**nr ows_c**
The number of rows of matrix **c**.

integer ::**ncols_c**
The number of columns of matrix **c**.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_hermitian_multiply_double**(**elpa_t** handle, **char** uplo_a, **char** uplo_c, **int** ncb, **datatype** *a, **datatype** *b, **int** nrows_b, **int** ncols_b, **datatype** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

char **uplo_a**;
: Should be set to 'U' if A is upper triangular, to 'L' if A is lower triangular or anything else if A is a full matrix.

char **uplo_c**;
: Should be set to 'U' if only the upper diagonal part of C is needed, to 'L' if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

int **ncb**;
: The number of columns of the global matrices **b** and **c**.

datatype ***a**;
: The host/device matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double". In case of a GPU b uild **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***b**;
: The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** must be "double". In case of a GPU b uild **b** can be a device pointer to a matrix **b** in the device memory.

int **nrows_b**;
: The number of rows of matrix **b**.

int **ncols_b**;
: The number of columns of matrix **b**.

datatype ***c**;
: The host/device matrix **c**. The dimensions of the matrix are specified by the parameters**n rows_c** and **ncols_c**. The**datatype** must be "double". In case of a GPU b uild **c** can be a device pointer to a matrix **c** in the device memory.

int **nrows_c**;
: The number of rows of matrix **c**.

int **ncols_c**;
: The number of columns of matrix **c**.

int ***error**;
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a "hermitian" multiplication C=A**T * B for real double-precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply_double** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.27 elpa_hermitian_multiply_double_complex

NAME

elpa_hermitian_multiply_double_complex − performs a "hermitian" multiplication of complex double-precision matrices: C = A**H * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**hermitian_multiply_double_complex** (uplo_a, uplo_c, ncb, a, b, nrows_b, ncols_b, &
c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

character*1 ::**uplo_a**
Should be set to 'U' if A is upper triangular, to 'L' if A is lower triangular or to anything else if A is a full matrix.

character*1 ::**uplo_c**
Should be set to 'U' if only the upper diagonal part of C is needed, to 'L' if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

integer ::**ncb**
The number of columns of the global matrices **b** and **c**.

datatype ::**a**
The host/device matrix **a**. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_double_complex)". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ::**b**
The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** of the matrix must be "comple x(kind=c_double_complex)". In case of a GPU build **b** can be a device pointer to a matrix **b** in the device memory.

integer ::**nr ows_b**
The number of rows of matrix **b**.

integer ::**ncols_b**
The number of columns of matrix **b**.

datatype ::**c**
The host/device matrix **c**. The dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. The**datatype** of the matrix must be "comple x(kind=c_double_complex)". In case of a GPU build **c** can be a device pointer to a matrix **c** in the device memory.

integer ::**nr ows_c**
The number of rows of matrix **c**.

integer ::**ncols_c**
The number of columns of matrix **c**.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_hermitian_multiply_double_complex**(**elpa_t** handle, **char** uplo_a, **char** uplo_c, **int** ncb, **datatype** *a, **datatype** *b, **int** nrows_b, **int** ncols_b, **datatype** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

char **uplo_a**;
Should be set to 'U' if A is upper triangular, to 'L' if A is lower triangular or anything else if A is a full matrix.

char **uplo_c**;
Should be set to 'U' if only the upper diagonal part of C is needed, to 'L' if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

int **ncb**;
The number of columns of the global matrices **b** and **c**.

datatype ***a**;
The host/device matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double comple x". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***b**;
The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** must be "double comple x". In case of a GPU build **b** can be a device pointer to a matrix **b** in the device memory.

int **nrows_b**;
The number of rows of matrix **b**.

int **ncols_b**;
The number of columns of matrix **b**.

datatype ***c**;
The host/device matrix**c**. The dimensions of the matrix are specified by the parameters**n r ows_c** and **ncols_c**. The**datatype** must be "double comple x". In case of a GPU build **c** can be a device pointer to a matrix **c** in the device memory.

int **nrows_c**;
The number of rows of matrix **c**.

int **ncols_c**;
The number of columns of matrix **c**.

int ***error**;
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a "hermitian" multiplication C=A**H * B for complex double-precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply_double_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.28 elpa_hermitian_multiply_float

NAME

elpa_hermitian_multiply_float − performs a "hermitian" multiplication of real single-precision matrices: C = A**T * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**hermitian_multiply_float** (uplo_a, uplo_c, ncb, a, b, nrows_b, ncols_b, &
c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

character*1 ::**uplo_a**
Should be set to ’U’ if A is upper triangular, to ’L’ if A is lower triangular or to anything else if A is a full matrix.

character*1 ::**uplo_c**
Should be set to ’U’ if only the upper diagonal part of C is needed, to ’L’ if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

integer ::**ncb**
The number of columns of the global matrices **b** and **c**.

datatype ::**a**
The host/device matrix **a**. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_float)". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ::**b**
The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** of the matrix must be "real(kind=c_float)". In case of a GPU b uild **b** can be a device pointer to a matrix **b** in the device memory.

integer ::**nr ows_b**
The number of rows of matrix **b**.

integer ::**ncols_b**
The number of columns of matrix **b**.

datatype ::**c**
The host/device matrix **c**. The dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. The**datatype** of the matrix must be "real(kind=c_float)". In case of a GPU b uild **c** can be a device pointer to a matrix **c** in the device memory.

integer ::**nr ows_c**
The number of rows of matrix **c**.

integer ::**ncols_c**
The number of columns of matrix **c**.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_hermitian_multiply_float**(**elpa_t** handle, **char** uplo_a, **char** uplo_c, **int** ncb, **datatype** *a, **datatype** *b, **int** nrows_b, **int** ncols_b, **datatype** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

char **uplo_a**;
: Should be set to ’U’ if A is upper triangular, to ’L’ if A is lower triangular or anything else if A is a full matrix.

char **uplo_c**;
: Should be set to ’U’ if only the upper diagonal part of C is needed, to ’L’ if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

int **ncb**;
: The number of columns of the global matrices **b** and **c**.

datatype ***a**;
: The host/device matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float". In case of a GPU b uild **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***b**;
: The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** must be "float". In case of a GPU b uild **b** can be a device pointer to a matrix **b** in the device memory.

int **nrows_b**;
: The number of rows of matrix **b**.

int **ncols_b**;
: The number of columns of matrix **b**.

datatype ***c**;
: The host/device matrix **c**. The dimensions of the matrix are specified by the parameters**n rows_c** and **ncols_c**. The**datatype** must be "float". In case of a GPU b uild **c** can be a device pointer to a matrix **c** in the device memory.

int **nrows_c**;
: The number of rows of matrix **c**.

int **ncols_c**;
: The number of columns of matrix **c**.

int ***error**;
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a "hermitian" multiplication C=A**T * B for real single-precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply_float** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.29 elpa_hermitian_multiply_float_complex

NAME

elpa_hermitian_multiply_float_complex − performs a "hermitian" multiplication of complex single-precision matrices: C = A**H * B

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**hermitian_multiply_float_complex** (uplo_a, uplo_c, ncb, a, b, nrows_b, ncols_b, &
c, nrows_c, ncols_c, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

character*1 ::**uplo_a**
Should be set to 'U' if A is upper triangular, to 'L' if A is lower triangular or to anything else if A is a full matrix.

character*1 ::**uplo_c**
Should be set to 'U' if only the upper diagonal part of C is needed, to 'L' if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

integer ::**ncb**
The number of columns of the global matrices **b** and **c**.

datatype ::**a**
The host/device matrix **a**. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_float_complex)". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ::**b**
The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** of the matrix must be "comple x(kind=c_float_complex)". In case of a GPU build **b** can be a device pointer to a matrix **b** in the device memory.

integer ::**nr ows_b**
The number of rows of matrix **b**.

integer ::**ncols_b**
The number of columns of matrix **b**.

datatype ::**c**
The host/device matrix **c**. The dimensions of the matrix are specified by the parameters **nrows_c** and **ncols_c**. The**datatype** of the matrix must be "comple x(kind=c_float_complex)". In case of a GPU build **c** can be a device pointer to a matrix **c** in the device memory.

integer ::**nr ows_c**
The number of rows of matrix **c**.

integer ::**ncols_c**
The number of columns of matrix **c**.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_hermitian_multiply_float_complex**(**elpa_t** handle, **char** uplo_a, **char** uplo_c, **int** ncb, **datatype** *a, **datatype** *b, **int** nrows_b, **int** ncols_b, **datatype** *c, **int** nrows_c, **int** ncols_c, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

char **uplo_a**;
: Should be set to 'U' if A is upper triangular, to 'L' if A is lower triangular or anything else if A is a full matrix.

char **uplo_c**;
: Should be set to 'U' if only the upper diagonal part of C is needed, to 'L' if only the upper diagonal part of C is needed, or to anything else if the full matrix C is needed.

int **ncb**;
: The number of columns of the global matrices **b** and **c**.

datatype ***a**;
: The host/device matrix **a**. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float comple x". In case of a GPU build **a** can be a device pointer to a matrix **a** in the device memory.

datatype ***b**;
: The host/device matrix **b**. The dimensions of the matrix are specified by the parameters **nrows_b** and **ncols_b**. The**datatype** must be "float comple x". In case of a GPU build **b** can be a device pointer to a matrix **b** in the device memory.

int **nrows_b**;
: The number of rows of matrix **b**.

int **ncols_b**;
: The number of columns of matrix **b**.

datatype ***c**;
: The host/device matrix**c**. The dimensions of the matrix are specified by the parameters**n r ows_c** and **ncols_c**. The**datatype** must be "float comple x". In case of a GPU build **c** can be a device pointer to a matrix **c** in the device memory.

int **nrows_c**;
: The number of rows of matrix **c**.

int **ncols_c**;
: The number of columns of matrix **c**.

int ***error**;
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

DESCRIPTION

Performs a "hermitian" multiplication C=A**H * B for complex single-precision matrices. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_hermitian_multiply_float_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_solve_tridiagonal**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.30 elpa_init

NAME

elpa_init − initializes the ELPA library

SYNOPSIS

FORTRAN INTERFACE

use elpa

error = **elpa_init** (api_version)

With the definitions of the input and output variables:

integer, intent(in) :: **api_version**

The api version that you want to initialize, currently the version is 20171201

integer ::**err or**

The return code. If the function returns without an error, the error code will be ELPA_OK.

C INTERFACE

#include <elpa/elpa.h>

int error = **elpa_init** (**int** api_version);

With the definitions of the input and output variables:

int **api_version**;

The api version that you want to initialize currently the version is 20171201

int **error**;

The return code. If the function returns without an error, the error code will be ELPA_OK.

DESCRIPTION

Initializes the ELPA library for usage. The return code should be ELPA_OK. The return code can be queried with the **elpa_strerr**(3) function.

SEE ALSO

elpa2_print_kernels(1) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.31 elpa_invert_triangular

NAME

elpa_invert_triangular − inverts an upper triangular matrix.

There are also variations of this routine that can accept not only host but also device pointers as input/output. Names of these routines explicitly contain the corresponding datatypes: elpa_invert_triangular_double, elpa_invert_triangular_float, elpa_invert_triangular_double_complex, elpa_invert_triangular_float_complex.

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**invert_triangular** (a, error)

With the definitions of the input and output variables:

datatype ::**a**
The matrix **a** that should be inverted. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)", "real(kind=c_float)", "complex(kind=c_double)", or "complex(kind=c_float)".

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_invert_triangular**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix that should be inverted. The dimensions of the matrix must be set*BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double", "float", "double complex", or "float complex".

int ***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular real or complex matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.32 elpa_invert_triangular_double

NAME

elpa_invert_triangular − inverts an upper triangular real double-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**invert_triangular_double** (a, error)

With the definitions of the input and output variables:

datatype :: **a** ! can also be a device pointer of type(c_ptr)

The host/device matrix **a** that should be inverted. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_double)". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

integer, optional :: **error**

The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_invert_triangular_double**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;

The handle to the ELPA object

datatype ***a**; // can also be a device pointer

The host/device matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular real double-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_double** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.33 elpa_invert_triangular_double_complex

NAME

elpa_invert_triangular − inverts an upper triangular complex double-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**invert_triangular_double_complex** (a, error)

With the definitions of the input and output variables:

datatype :: **a** ! can also be a device pointer of type(c_ptr)

The host/device matrix **a** that should be inverted. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_double)". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

integer, optional :: **error**

The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_invert_triangular_double_complex**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;

The handle to the ELPA object

datatype ***a**; // can also be a device pointer

The host/device matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "double complex". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular complex double-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_double_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.34 elpa_invert_triangular_float

NAME

elpa_invert_triangular − inverts an upper triangular real single-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**invert_triangular_float** (a, error)

With the definitions of the input and output variables:

datatype :: **a** ! can also be a device pointer of type(c_ptr)

The host/device matrix **a** that should be inverted. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "real(kind=c_float)". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

integer, optional :: **error**

The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_invert_triangular_float**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;

The handle to the ELPA object

datatype ***a**; // can also be a device pointer

The host/device matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular real single-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_float** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.35 `elpa_invert_triangular_float_complex`

NAME

elpa_invert_triangular − inverts an upper triangular complex single-precision matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**invert_triangular_float_complex** (a, error)

With the definitions of the input and output variables:

datatype :: **a** ! can also be a device pointer of type(c_ptr)
: The host/device matrix **a** that should be inverted. The dimensions of matrix**a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix must be "complex(kind=c_float)". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

integer, optional :: **error**
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_invert_triangular_float_complex**(**elpa_t** handle, **datatype** *a, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

datatype ***a**; // can also be a device pointer
: The host/device matrix that should be inverted. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** must be "float complex". In case of a GPU build **a** can be a device pointer of type "type(c_ptr)" to matrix **a** in the device memory.

int ***error**;
: The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Inverts an upper triangular complex single-precision matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_invert_triangular_float_complex** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_choleksy**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.36 elpa_load_settings

NAME

elpa_load_settings − loads the setting of an elpa object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**load_settings** (filename, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object

character(*) ::**filename**
The file from where to load the settings

integer, optional :: **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_load_settings**(**elpa_t** handle, **const char** *filename, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

const char ***filename**;
The filename to load the settings

int ***error**;
The error return code

DESCRIPTION

Loads all the settings of an previously stored ELPA object from a file specified via the **filename** parameter.

SEE ALSO

elpa_store_setting(3)

C.37 elpa_print_settings

NAME

elpa_print_settings − prints the setting of an elpa object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**print_settings** (error)

With the definitions of the input and output variables:

class(elpa_t) **elpa**
An instance of the ELPA object

integer, optional **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_print_settings**(**elpa_t** handle, **int** *error):

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

int ***error**;
The error return code

DESCRIPTION

Prints all the settings of an ELPA object. The settings can be stored, or loaded with **elpa_store_settings**.3 or **elpa_load_settings**.3

SEE ALSO

elpa_store_setting(3) **elpa_load_settings**.(3)

C.38 elpa_set

NAME

elpa_set − set parameter or tunables for the ELPA library

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**set** (**character(*)** name, **datatype** value, **integer** error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object.

character(*) ::**name**
the name of the option to be set

datatype ::**v alue**
the value which should be assigned to the option **name**. The**datatype** can be **integer** or **real(kind=c_double)**.

integer, optional :: **error**
The returned error code. On success it is ELPA_OK, otherwise an error. he error code can be queried with **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void elpa_set (**elpa_t** handle, **const char** *name, **datatype** value, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

const char ***name**;
The name of the option to be set.

datatype **value**;
The value which should be assigned to the option **name**. The**datatype** can be either **int** or **double**.

DESCRIPTION

The **elpa_set** function is used to set **mandatory parameters** and **runtime options** of the ELPA library. It returns an error code which can be queried with **elpa_strerr**(3).

Mandatory parameters:

Mandatory parameters of an ELPA instance have to be set *BEFORE* the ELPA instance is set up with the function **elpa_setup**(3).

At the moment the following mandatory parameters are supported:

"na": integer parameter. The global matrix has size is (na * na)

"nev": integer parameter. The number of eigenvectors to be computed in a call to **elpa_eigenvectors**(3). Must satisfy 1 <= nev <= na.

"local_nrows":
integer parameter. Number of matrix rows stored on this MPI process.

"local_ncols":
integer parameter. Number of matrix columns stored on this MPI process.

"process_row":
integer parameter. Process row number in the 2D domain decomposition.

"process_col":
integer parameter. Process column number in the 2D domain decomposition.

"mpi_comm_parent":
integer parameter. The parent MPI communicator which includes all MPI process which are used in the 2D domain decomposition.

"bandwidth":
integer parameter. Some ELPA computational steps can be accelerated if the input matrix is already in banded form. If set, ELPA assumes that the matrix has the provided bandwidth.

"BLACS_context":
integer parameter. The generalized eigenvalue solver **elpa_generalized_eigenvectors**(3) uses internal calls to some of the ScaLAPACK routines. Thus before calling it, the user has to provide properly initialized BLACS context.

"timings":
integer parameter. Choose whether time measurements should be done in the ELPA routines (1) or not (0).

Runtime options:

Runtime options of an ELPA option can be set at *any time*.

At the moment the following runtime options are supported:

"solver":
Choose which solver should be used in the compute steps **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3). At the moment allowed option are **"ELPA_SOLVER_1STAGE"** or **"ELPA_SOLVER_2STAGE"**.

"real_kernel":
Choose which real kernel should be used in the **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3) compute steps, if solver is set to **"ELPA_SOLVER_2STAGE"**. The available kernels can be queried with **elpa2_print_kernels**(1).

"complex_kernel":
Choose which complex kernel should be used in the **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3) compute steps, if solver is set to **"ELPA_SOLVER_2STAGE"**. The available kernels can be queried with **elpa2_print_kernels**(1).

"qr": Choose whether a QR decomposition should be used for the real case computations in **elpa_eigenvalues**(3) or **elpa_eigenvectors**(3) computational steps, if solver was set to **"ELPA_SOLVER_2STAGE"**.

"gpu": Choose whether accelerated GPU calculations should be used. Only available if ELPA has been build with GPU support.

"debug":
Choose whether, in case of an error, more debug information should be provided.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_deallocate**(3) **elpa_uninit**(3)

C.39 elpa_setup

NAME

elpa_setup − setup an instance of the ELPA library

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

success = elpa%**setup**()

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

integer :: **success**
The returned error code. Should normally be ELPA_OK. Can be queried with **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

int success = **elpa_setup** (**elpa_t** handle);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle of an ELPA object, obtained before with **elpa_allocate**(3)

int **success**;
The returned error code. Should normally be ELPA_OK. Can be queried with **elpa_strerr**(3)

DESCRIPTION

Setups an ELPA object. *Prior* to calling the setup, the functions **elpa_init**(3), **elpa_allocate**(3) *must have been called* and some parameters must have been set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_deallocate**(3) **elpa_uninit**(3)

C.40 elpa_skew_eigenvalues

NAME

elpa_skew_eigenvalues − computes all eigenvalues of a real skew-symmetric matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**skew_eigenvalues** (a, ev, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object.

datatype :: **a**
The matrix **a** for which the eigenvalues should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)" or "real(kind=c_float)". The matrix has to be skew-symmetric, this is not checked by the routine.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix.

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3).

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_skew_eigenvalues**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "double" or "float". The matrix has to be skew-symmetric, this is not checked by the routine.

datatype ***ev**;
The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float".

int ***error**;
The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Computes the eigenvalues of a real skew-symmetric matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_skew_eigenvalues** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvectors**(3) **elpa_skew_eigenvectors**(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_eigenvalues**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.41 elpa_skew_eigenvectors

NAME

elpa_skew_eigenvectors − computes the eigenvalues and (part of) the eigenvector spectrum for a real skew-symmetric matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**skew_eigenvectors** (a, ev, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
An instance of the ELPA object

datatype :: **a**
The matrix **a** for which the eigenvalues and eigenvectors should be computed. The dimensions of matrix **a** must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** of the matrix can be one of "real(kind=c_double)" or "real(kind=c_float)". The matrix has to be skew-symmetric, this is not checked by the routine.

datatype :: **ev**
The vector **ev** where the eigenvalues will be stored in *ascending* order. The**datatype** of the v ector **ev** can be either "real(kind=c_double)" or "real(kind=c_float)", depending of the **datatype** of the matrix.

datatype :: **q**
The storage space for the computed eigenvectors. The number of requested eigenpairs must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be one of "complex(kind=c_double)" or "complex(kind=c_float)". Note, that for a skew-symmetric matrix the eigenvectors are complex. The routine returns separately the real and imaginary parts of the complex eigenvectors. Thus, the storage space has to be of dimension q(#number_of_rows, 2*#number_of_columns).

integer, optional :: **error**
The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function fB elpa_strerr(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_eigenvalues**(**elpa_t** handle, **datatype** *a, **datatype** *ev, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

datatype ***a**;
The matrix **a** for which the eigenvalues and eigenvectors should be computed. The dimensions of the matrix must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can be "double" or "float". The matrix has to be symmetric or hermitian, this is not checked by the routine.

datatype ***ev**;

The storage for the computed eigenvalues. Eigenvalues will be stored in *ascending* order. The **datatype** can be either "double" or "float".

datatype ***q**;

The storage space for the computed eigenvectors. The number of requested eigenvectors must be set *BEFORE* with the methods **elpa_set**(3) and **elpa_setup**(3). The**datatype** can "double complex" or "float complex". Note, that for a skew-symmetric matrix the eigenvectors are complex. The routine returns separately the real and imaginary parts of the complex eigenvectors. Thus, the storage space has to be of dimension q(#number_of_rows, 2*#number_of_columns).

int ***error**;

The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3).

DESCRIPTION

Computes the eigenvalues and (part of) the eigenvector spectrum of a real symmetric or complex hermitian matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_skew_eigenvectors** can be called. In particular, the number of the requested eigenpairs, "nev", must be set with **elpa_set**(3).

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_skew_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.42 elpa_solve_tridiagonal

NAME

elpa_solve_tridiagonal − computes the eigenvalue problem for real symmetric tridiagonal matrix

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**solve_tridiagonal** (d, e, q, error)

With the definitions of the input and output variables:

class(elpa_t) :: **elpa**
: An instance of the ELPA object.

datatype :: **d**
: The diagonal elements of a matrix whose dimensions have been defined in **elpa_setup**(3). The dimensions of the matrix must be set *BEFORE* with **elpa_setup**(3). On exit the eigenvalues are stored in **d**. The**datatype** of the diagonal elements can either be "real(kind=c_double)" or "real(kind=c_float)".

datatype :: **e**
: The offdiagonal elements of the matrix. The **datatype** of the diagonal elements can either be "real(kind=c_double)" or "real(kind=c_float)".

datatype :: **q**
: The storage space for the computed eigenvectors. The**datatype** of the matrix can be either "real(kind=c_double)" or "real(kind=c_float)".

integer, optional :: **error**
: The return error code of the function. Should be "ELPA_OK". The error code can be queried with the function **elpa_strerr**(3)

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_solve_tridiagonal**(**elpa_t** handle, **datatype** *d, **datatype** *e, **datatype** *q, **int** *error);

With the definitions of the input and output variables:

elpa_t **handle**;
: The handle to the ELPA object

datatype ***d**;
: The diagonal elements of the matrix. The dimensions of the matrix must be set *BEFORE* with **elpa_setup**(3). On exit the eigenvalues are stored in **d**. The**datatype** can be one of "double" or "float".

datatype ***e**;
: The offdiagonal elements of the matrix. The **datatype** can be one of "double" or "float".

datatype ***q**;
: The storage space for the computed eigenvectors. The **datatype** can be one of "double" or "float".

int ***error**;
: The error code of the function. Should be "ELPA_OK". The error codes can be queried with **elpa_strerr**(3)

DESCRIPTION

Computes the eigenvalue problem of a real symmetric tridiagonal matrix. The functions **elpa_init**(3), **elpa_allocate**(3), **elpa_set**(3), and **elpa_setup**(3) must be called *BEFORE* **elpa_solve_tridiagonal** can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_setup**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_hermitian_multiply**(3) **elpa_uninit**(3) **elpa_deallocate**(3)

C.43 elpa_store_settings

NAME

elpa_store_settings − stores the setting of an elpa object

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call elpa%**store_settings** (filename, error)

With the definitions of the input and output variables:

class(elpa_t) ::**elpa**
An instance of the ELPA object

character(*) ::**filename**
The filename to be used for storing the settings

integer, optional :: **error**
An error return code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void **elpa_store_settings**(**elpa_t** handle, **const char** *filename, **int** *error):

With the definitions of the input and output variables:

elpa_t **handle**;
The handle to the ELPA object

const char ***filename**;
The filename to store the settings

int ***error**;
The error return code

DESCRIPTION

Stores all the settings of an ELPA object in a human readable form to a file specified via the **filename** parameter. The settings can later be restored with the **elpa_load_settings**(3) method.

SEE ALSO

elpa_load_setting(3)

C.44 elpa_uninit

NAME

elpa_uninit − uninitializes the ELPA library

SYNOPSIS

FORTRAN INTERFACE

use elpa
class(elpa_t), pointer :: elpa

call **elpa_uninit** (error)

With the definitions of the input and output variables:

integer, optional :: error
The error code

C INTERFACE

#include <elpa/elpa.h>
elpa_t handle;

void elpa_uninit (int *error);

With the definitions of the input and output variables:

int **error***;
The error code

DESCRIPTION

Uninitializes the ELPA library after usage. The function **elpa_init**(3) must have been called *BEFORE* elpa_uninit can be called.

SEE ALSO

elpa2_print_kernels(1) **elpa_init**(3) **elpa_allocate**(3) **elpa_set**(3) **elpa_strerr**(3) **elpa_eigenvalues**(3) **elpa_eigenvectors**(3) **elpa_cholesky**(3) **elpa_invert_triangular**(3) **elpa_solve_tridiagonal**(3) **elpa_hermitian_multiply**(3) **elpa_setup**(3) **elpa_deallocate**(3)