XCONFIGURE

XCONFIGURE is a collection of configure wrapper scripts for various HPC applications. The purpose of the scripts is to configure the application in question to make use of Intel's software development tools (Intel Compiler, Intel MPI, Intel MKL). XCONFIGURE helps to rely on a "build recipe", which is known to expose the highest performance or to reliably complete the build process.

> Contributions are very welcome!

Each application (or library) is hosted in a separate directory. To configure (and ultimately build) an application, one can rely on a single script which then downloads a specific wrapper into the current working directory (of the desired application).

```
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh qe hsw
```

On systems without access to the Internet, one can download (or clone) the entire collection upfront. To configure an application, please open the "config" folder and follow the build recipe of the desired application.

Documentation

- **ReadtheDocs**: online documentation with full text search.
- **PDF**: a single documentation file.

Related Projects

- Spack Package Manager: http://computation.llnl.gov/projects/spack-hpc-package-manager
- EasyBuild / EasyConfig (University of Gent): https://github.com/easybuilders

Please note that XCONFIGURE has a narrower scope when compared to the above package managers.

Applications

CP2K

This document focuses on building and running the Intel fork of CP2K. The fork was formerly a branch of CP2K's Git-mirror; CP2K is meanwhile natively hosted at GitHub. This work is supposed to track the master version of CP2K in a timely fashion. The LIBXSMM library is highly recommended and can be found at https://github.com/hfp/libxsmm. In terms of functionality (and performance) it is beneficial to rely on LIBINT and LIBXC, whereas ELPA eventually improves the performance. For high performance, LIBXSMM has been incorporated since CP2K 3.0 (and intends to substitute CP2K's "libsmm" library).

Below are the releases of the Intel Compiler, which are known to reproduce correct results according to the regression tests (it is possible to combine components from different versions):

- Intel Compiler 2017 (u0, u1, u2, u3), *and* the **initial** release of MKL 2017 (u0)
  - source /opt/intel/compilers_and_libraries_2017.[*u0-u3*]/linux/bin/compilervars.sh intel64
    source /opt/intel/compilers_and_libraries_2017.0.098/linux/mkl/bin/mklvars.sh intel64
- Intel Compiler 2017 Update 4, and any later update of the 2017 suite (u4, u5, u6, u7)
  - source /opt/intel/compilers_and_libraries_2017.[*u4-u7*]/linux/bin/compilervars.sh intel64
- Intel Compiler 2018 (u3, u4, u5): only with CP2K/development (not with CP2K 6.1 or earlier)
  - source /opt/intel/compilers_and_libraries_2018.3.222/linux/bin/compilervars.sh intel64
  - source /opt/intel/compilers_and_libraries_2018.5.274/linux/bin/compilervars.sh intel64

- Intel Compiler 2019 (u1, u2, u3): failure at runtime
- Intel MPI; usually any version is fine

There are no configuration wrapper scripts provided for CP2K, please follow below recipe. However, attempting to run below command yields an info-script:

```
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh cp2k
```

Of course, the above can be simplified:

```
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/config/cp2k/info.sh
chmod +x info.sh
```

Build Instructions

Build the Intel-fork of CP2K To build CP2K/Intel from source, one may rely on Intel Compiler 16, 17, or 18 series:

```
source /opt/intel/compilers_and_libraries_2018.3.222/linux/bin/compilervars.sh intel64
```

LIBXSMM is automatically built in an out-of-tree fashion when building CP2K/Intel fork. The only prerequisite is that the LIBXSMMROOT path needs to be detected (or supplied on the `make` command line). LIBXSMMROOT is automatically discovered automatically if it is in the user's home directory, or when it is in parallel to the CP2K directory. By default (no `AVX` or `MIC` is given), the build process is carried out using the `-xHost` target flag. For example, to explicitly target "Skylake" (SKX):

```
git clone https://github.com/hfp/libxsmm.git
git clone https://github.com/hfp/cp2k.git
cd cp2k; rm -rf exe lib obj
make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=3 MIC=0
```

Most if not all hot-spots in CP2K are covered by libraries (e.g., LIBXSMM). It can be beneficial to rely on the GNU Compiler tool-chain. To only use Intel libraries such as Intel MPI and Intel MKL, one can rely on the GNU-key (`GNU=1`):

```
git clone https://github.com/hfp/libxsmm.git
git clone https://github.com/hfp/cp2k.git
cd cp2k; rm -rf exe lib obj
make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=3 MIC=0 GNU=1
```

Using the GNU tool-chain requires to configure LIBINT, LIBXC, and ELPA accordingly (e.g., `configure-elpa-skx-gnu-omp.sh` instead of `configure-elpa-skx-omp.sh`). To further adjust CP2K at build time, additional key-value pairs can be passed at Make's command line (like `ARCH=Linux-x86-64-intelx` or `VERSION=psmp`).

- **SYM**: set `SYM=1` to include debug symbols into the executable e.g., helpful with performance profiling.
- **DBG**: set `DBG=1` to include debug symbols, and to generate non-optimized code.

To further improve performance and versatility, one should supply LIBINTROOT, LIBXCROOT, and ELPAROOT. These keys are valid when relying on CP2K/Intel's ARCH files (see later sections about these libraries).

Build an Official Release Here are two ways to build an official release of CP2K using an Intel tool chain:

- Use the ARCH files from CP2K/intel fork.
- Write an own ARCH file.

NOTE: the step-by-step guide builds CP2K 6.1.0 with LIBXSMM, ELPA, LIBXC, and LIBINT.

LIBXSMM is supported since CP2K 3.0. CP2K 6.1 includes `Linux-x86-64-intel.*` (`arch` directory) as a starting point for writing an own ARCH-file (note: `Linux-x86-64-intel.*` vs. `Linux-x86-64-intelx.*`). Remember, performance critical code is often located in libraries (hence `-O2` optimizations for CP2K's source code are sufficient in almost all cases), more important for performance are target-flags such as `-march=native` (`-xHost`) or `-mavx2 -mfma`. Prior to Intel Compiler 2018, the flag `-fp-model source` (FORTRAN) and `-fp-model precise` (C/C++) are key for passing CP2K's regression tests. Please follow the official guide and consider the CP2K Forum in case of trouble. If an own ARCH file is used or prepared, the LIBXSMM library needs to be built separately. Building LIBXSMM is rather simple; to build the master revision:

```
git clone https://github.com/hfp/libxsmm.git
cd libxsmm ; make
```

To build an official release:

```
wget https://github.com/hfp/libxsmm/archive/1.9.tar.gz
tar xvf 1.9.tar.gz
cd libxsmm-1.9 ; make
```

Taking the ARCH files that are part of the CP2K/Intel fork automatically picks up the correct paths for Intel libraries. These paths are determined by using the environment variables setup when the Intel tools are source'd. Similarly, LIBXSMMROOT (which can be supplied on Make's command line) is discovered automatically if it is in the user's home directory, or when it is in parallel to the CP2K directory (as demonstrated below).

```
git clone https://github.com/hfp/libxsmm.git
wget https://github.com/cp2k/cp2k/archive/v6.1.0.tar.gz
tar xvf v6.1.0.tar.gz
```

To download the ARCH files from the Intel-fork, simply run the following:

```
cd cp2k-6.1.0
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh cp2k
```

Alternatively, one can download the afore mentioned ARCH-files manually:

```
cd cp2k-6.1.0/arch
wget https://github.com/hfp/cp2k/raw/master/arch/Linux-x86-64-intelx.arch
wget https://github.com/hfp/cp2k/raw/master/arch/Linux-x86-64-intelx.popt
wget https://github.com/hfp/cp2k/raw/master/arch/Linux-x86-64-intelx.psmp
wget https://github.com/hfp/cp2k/raw/master/arch/Linux-x86-64-intelx.sopt
wget https://github.com/hfp/cp2k/raw/master/arch/Linux-x86-64-intelx.ssmp
```

To build the official CP2K sources/release now works the same way as for the Intel-fork:

```
source /opt/intel/compilers_and_libraries_2018.3.222/linux/bin/compilervars.sh intel64
cd cp2k-6.1/makefiles; make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=2
```

To further improve performance and versatility, one may supply LIBINTROOT, LIBXCROOT, and ELPAROOT when relying on CP2K/Intel's ARCH files (see the following section about these libraries).

LIBINT, LIBXC, and ELPA To configure, build, and install LIBINT (version 1.1.5 and 1.1.6 have been tested), one can proceed with https://xconfigure.readthedocs.io/libint/. Also note there is no straightforward way to cross-compile LIBINT 1.1.x for an instruction set extension that is not supported by the compiler host. To incorporate LIBINT into CP2K, the key `LIBINTROOT=/path/to/libint` needs to be supplied when using CP2K/Intel's ARCH files (make).

To configure, build, and install LIBXC (version 3.0.0 has been tested), and one can proceed with https://xconfigure.readthedo
To incorporate LIBXC into CP2K, the key `LIBXCROOT=/path/to/libxc` needs to be supplied when using CP2K/Intel's ARCH files (make). After CP2K 5.1, only the latest major release of LIBXC (by the time of the CP2K-release) will be supported (e.g., LIBXC 4.x by the time of CP2K 6.1).

To configure, build, and install the Eigenvalue SoLvers for Petaflop-Applications (ELPA), one can proceed with https://xconfigure.readthedocs.io/libint/. To incorporate ELPA into CP2K, the key `ELPAROOT=/path/to/elpa` needs to be supplied when using CP2K/Intel's ARCH files (make). The Intel-fork defaults to ELPA-2017.11 (earlier versions can rely on the ELPA key-value pair e.g., `ELPA=201611`).

```
make ARCH=Linux-x86-64-intelx VERSION=psmp ELPAROOT=/path/to/elpa/default-arch
```

At runtime, a build of the Intel-fork supports an environment variable CP2K_ELPA:

- **CP2K_ELPA=-1**: requests ELPA to be enabled; the actual kernel type depends on the ELPA configuration.
- **CP2K_ELPA=0**: ELPA is not enabled by default (only on request via input file); same as non-Intel fork.
- **CP2K_ELPA**=<not-defined>: requests ELPA-kernel according to CPUID (default with CP2K/Intel-fork).

Step-by-step Guide This step-by-step guide aims to build an MPI/OpenMP-hybrid version of the official release of CP2K using the GNU Compiler Collection, Intel MPI, Intel MKL, LIBXSMM, ELPA, LIBXC, and LIBINT. Internet connectivity is assumed on the build-system. Please note that such limitations can be worked around or avoided with additional steps. However, this simple step-by-step guide aims to make some reasonable assumptions.

As the step-by-step guide uses GNU Fortran, only Intel MPI and Intel MKL are sourced (sourcing all Intel development tools does not harm). Regarding GNU Fortran, version 7.x or 8.x is recommended (older versions may not be sufficient).

```
source /opt/intel/compilers_and_libraries_2018.5.274/linux/mpi/intel64/bin/mpivars.sh
source /opt/intel/compilers_and_libraries_2019.3.199/linux/mkl/bin/mklvars.sh intel64
```

The first step builds ELPA. Do not use an ELPA-version newer than 2017.11.001.

```
cd $HOME
wget https://elpa.mpcdf.mpg.de/html/Releases/2017.11.001/elpa-2017.11.001.tar.gz
tar xvf elpa-2017.11.001.tar.gz
cd elpa-2017.11.001
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh elpa
./configure-elpa-skx-gnu-omp.sh
make -j
make install
make clean
```

The second step builds LIBINT, which should not be cross-compiled. Simply compile on the real target-architecture.

```
cd $HOME
wget --no-check-certificate https://github.com/evaleev/libint/archive/release-1-1-6.tar.gz
tar xvf release-1-1-6.tar.gz
cd libint-release-1-1-6
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh libint
configure-libint-skx-gnu.sh
make -j
make install
make distclean
```

The third step builds LIBXC.

```
cd $HOME
wget --content-disposition http://www.tddft.org/programs/octopus/down.php?file=libxc/4.3.4/libxc-4.3.4
tar xvf libxc-4.3.4.tar.gz
cd libxc-4.3.4
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh libxc
configure-libxc-skx-gnu.sh
make -j
make install
make distclean
```

The fourth step makes LIBXSMM available, which is compiled as part of the next step.

```
cd $HOME
wget https://github.com/hfp/libxsmm/archive/master.tar.gz
tar xvf libxsmm-master.tar.gz
```

This last step builds the PSMP-variant of CP2K. Please re-download the ARCH-files from GitHub as mentioned below (do not reuse older/outdated files).

```
cd $HOME
wget https://github.com/cp2k/cp2k/archive/v6.1.0.tar.gz
tar xvf cp2k-6.1.0.tar.gz
cd cp2k-6.1.0
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh cp2k
patch -p0 src/pw/fft/fftw3_lib.F intel-mkl.diff
rm -rf exe lib obj
cd makefiles
make ARCH=Linux-x86-64-intelx VERSION=psmp GNU=1 AVX=3 MIC=0 \
  LIBINTROOT=$HOME/libint/gnu-skx \
  LIBXCROOT=$HOME/libxc/gnu-skx \
  ELPAROOT=$HOME/elpa/gnu-skx-omp -j
```

The CP2K executable should be now ready (`exe/Linux-x86-64-intelx/cp2k.psmp`).

Memory Allocation Dynamic allocation of heap memory usually requires global book keeping eventually incurring overhead in shared-memory parallel regions of an application. For this case, specialized allocation strategies are available. To use such a strategy, memory allocation wrappers can be used to replace the default memory allocation at build-time or at runtime of an application.

To use the malloc-proxy of the Intel Threading Building Blocks (Intel TBB), rely on the `TBBMALLOC=1` key-value pair at build-time of CP2K (default: `TBBMALLOC=0`). Usually, Intel TBB is already available when sourcing the Intel development tools (one can check the TBBROOT environment variable). To use TCMALLOC as an alternative, set `TCMALLOCROOT` at build-time of CP2K by pointing to TCMALLOC's installation path (configured per `./configure --enable-minimal --prefix=<TCMALLOCROOT>`).

Run Instructions Running CP2K may go beyond a single node, and pinning processes and threads becomes even more important. There are several scheme available. As a rule of thumb, a high rank-count for lower node-counts may yield best results unless the workload is very memory intensive. In the latter case, lowering the number of MPI-ranks per node is effective especially if a larger amount of memory is replicated rather than partitioned by the rank-count. In contrast (communication bound), a lower rank count for multi-node computations may be desired. Most important, CP2K prefers a total rank-count to be a square-number (two-dimensional communication pattern) rather than a Power-of-Two (POT) number. This property can be as dominant as wasting cores per node is more effective than fully utilizing the entire node (sometimes a frequency upside over an "all-core turbo" emphasizes this property further). Counter-intuitively, even an unbalanced rank-count per node i.e., different rank-counts per socket can be an advantage.

Because of the above mentioned complexity, a script for planning MPI-execution (`plan.sh`) is available. Here is a first example for running the PSMP-binary i.e., MPI/OpenMP-hybrid CP2K on an HT-enabled dual-socket system with 24 cores per processor/socket (96 hardware threads). A first step would be to run with 48 ranks and 2 threads per core. However, a second try could execute 16 ranks with 6 threads per rank (`16x6`):

```
mpirun -np 16 \
  -genv I_MPI_PIN_DOMAIN=auto -genv I_MPI_PIN_ORDER=bunch \
  -genv OMP_PLACES=threads -genv OMP_PROC_BIND=SPREAD \
  -genv OMP_NUM_THREADS=6 \
  exe/Linux-x86-64-intelx/cp2k.psmp workload.inp
```

It is recommended to set `I_MPI_DEBUG=4`, which displays/logs the pinning and thread affinization (with no performance penalty) at startup of the application. The recommended `I_MPI_PIN_ORDER=bunch` ensures that ranks per node are split as even as possible with respect to sockets e.g., running 36 ranks on a 2x20-core system puts 2x18 ranks (instead of 20+16 ranks). To plan for running on 8 nodes (with above mentioned 48-core systems) may look like:

```
./plan.sh 8 48
================================================================================
384 cores: 8 node(s) with 2x24 core(s) per node and 2 threads per core
================================================================================
[48x2]: 48 ranks per node with 2 thread(s) per rank (6% penalty)
[24x2]: 24 ranks per node with 4 thread(s) per rank (6% penalty)
[12x2]: 12 ranks per node with 8 thread(s) per rank (6% penalty)
--------------------------------------------------------------------------------
[8x12]: 8 ranks per node with 12 thread(s) per rank (0% penalty) -> 8x8
--------------------------------------------------------------------------------
Try also 3 and 12 nodes!
```

The script (`plan.sh <num-node> <num-cores-per-node> <num-threads-per-core> <num-sockets>`) displays MPI/OpenMP configurations sorted by increasing waste of compute due to suiting the square-number preference (except for the first group where potential communication overhead is shown). For the first setup that suits the square-number preference (`24x4`), the MPI command line may look like:

```
mpirun -perhost 24 -host node1,node2,node3,node4,node5,node6,node7,node8 \
  -genv I_MPI_PIN_DOMAIN=auto -genv I_MPI_PIN_ORDER=bunch \
  -genv OMP_PLACES=threads -genv OMP_PROC_BIND=SPREAD \
  -genv OMP_NUM_THREADS=4 -genv I_MPI_DEBUG=4 \
  exe/Linux-x86-64-intelx/cp2k.psmp workload.inp
```

Please note that `plan.sh` stores the given arguments (except for the node-count) as default for the next plan (`$HOME/.xconfigure-cp2k-plan`). This allows to supply the system-type once, and to plan with varying node-counts in a convenient fashion.

Sanity Check There is nothing that can replace the full regression test suite. However, to quickly check whether a build is sane or not, one can run for instance `tests/QS/benchmark/H2O-64.inp` and check if the SCF iteration prints like the following:

```
  Step     Update method       Time    Convergence         Total energy    Change
  ------------------------------------------------------------------------------
     1 OT DIIS     0.15E+00    0.5     0.01337191     -1059.6804814927 -1.06E+03
     2 OT DIIS     0.15E+00    0.3     0.00866338     -1073.3635678409 -1.37E+01
     3 OT DIIS     0.15E+00    0.3     0.00615351     -1082.2282197787 -8.86E+00
     4 OT DIIS     0.15E+00    0.3     0.00431587     -1088.6720379505 -6.44E+00
     5 OT DIIS     0.15E+00    0.3     0.00329037     -1092.3459788564 -3.67E+00
     6 OT DIIS     0.15E+00    0.3     0.00250764     -1095.1407783214 -2.79E+00
     7 OT DIIS     0.15E+00    0.3     0.00187043     -1097.2047924571 -2.06E+00
     8 OT DIIS     0.15E+00    0.3     0.00144439     -1098.4309205383 -1.23E+00
     9 OT DIIS     0.15E+00    0.3     0.00112474     -1099.2105625375 -7.80E-01
    10 OT DIIS     0.15E+00    0.3     0.00101434     -1099.5709299131 -3.60E-01
    [...]
```

The column called "Convergence" must monotonically converge towards zero.

Performance The script for planning MPI-execution (`plan.sh`) is highly recommend along with reading the section about how to run CP2K. As soon as several experiments finished, it becomes handy to summarize the log-output. For this use case, an info-script (`info.sh`) is available attempting to present a table (summary of all results), which is generated from log files (use `tee`, or rely on the output of the job scheduler). There are only certain file extensions supported (`.txt`, `.log`). If no file matches, then all files (independent of the file extension) are attempted to be parsed (which will go wrong eventually). If for some reason the command to launch CP2K is not part of the log and the run-arguments cannot be determined otherwise, the number of nodes is eventually parsed using the filename of the log itself (e.g., first occurrence of a number along with an optional "n" is treated as the number of nodes used for execution).

```
./run-cp2k.sh | tee cp2k-h2o64-2x32x2.txt
ls -1 *.txt
cp2k-h2o64-2x32x2.txt
cp2k-h2o64-4x16x2.txt

./info.sh [-best] /path/to/logs-or-cwd
H2O-64             Nodes R/N T/R Cases/d Seconds
cp2k-h2o64-2x32x2 2      32   4     807 107.237
cp2k-h2o64-4x16x2 4      16   8     872  99.962
```

Please note that the number of cases per day (Cases/d) are currently calculated with integer arithmetic and eventually lower than just rounding down (based on 86400 seconds per day). The number of seconds taken are end-to-end (wall time), i.e. total time to solution including any (sequential) phase (initialization, etc.). Performance is higher if the workload requires more iterations (some publications present a metric based on iteration time).

References https://nholmber.github.io/2017/04/cp2k-build-cray-xc40/
https://www.cp2k.org/howto:compile

ELPA

Build Instructions

ELPA 2018.05.001 and 2018.11.001 Download and unpack ELPA and make the configure wrapper scripts available in ELPA's root folder. It is recommended to package the state (Tarball or similar), which is achieved after downloading the wrapper scripts.

NOTE: both 2018-versions **fail or crash in several regression tests** in CP2K (certain rank-counts produce an incorrect decomposition), and hence they should be avoided in production with CP2K or Quantum Espresso (QE).

```
wget https://elpa.mpcdf.mpg.de/html/Releases/2018.05.001/elpa-2018.05.001.tar.gz
tar xvf elpa-2018.05.001.tar.gz
cd elpa-2018.05.001
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh elpa
```

Please make the Intel Compiler and Intel MKL available on the command line. This depends on the environment. For instance, many HPC centers rely on `module load`.

```
source /opt/intel/compilers_and_libraries_2018.3.222/linux/bin/compilervars.sh intel64
```

For example, to configure and make for an Intel Xeon Scalable processor ("SKX"):

```
make clean
./configure-elpa-skx-omp.sh
make -j ; make install

make clean
./configure-elpa-skx.sh
make -j ; make install
```

After building and installing the desired configuration(s), one may have a look at the installation:

```
[user@system elpa-2018.05.001]$ ls ../elpa
 default-skx
 default-skx-omp
```

For different targets (instruction set extensions) or for different versions of the Intel Compiler, the configure scripts support an additional argument ("default" is the default tagname):

```
./configure-elpa-skx-omp.sh tagname
```

As shown above, an arbitrary "tagname" can be given (without editing the script). This might be used to build multiple variants of the ELPA library.

ELPA 2017.11.001 (and older) Download and unpack ELPA and make the configure wrapper scripts available in ELPA's root folder. It is recommended to package the state (Tarball or similar), which is achieved after downloading the wrapper scripts.

NOTE: this version of ELPA must be used with Quantum Espresso's ___ELPA__2018 interface (`-D__ELPA_2018`), which is patched into QE by default when using XCONFIGURE's up-to-date build wrapper scripts. The ___ELPA__2017 preprocessor definition triggers the ELPA1 legacy interface (get_elpa_row_col_comms, etc.), which was removed after ELPA 2017.05.003. Also, it appears `make clean` (or similar Makefile target) for ELPA 2016.11.001 is cleaning up the entire directory including all "non-ELPA content" (the directory also remains somewhat unclean such that subsequent builds may fail)

```
wget https://elpa.mpcdf.mpg.de/html/Releases/2017.11.001/elpa-2017.11.001.tar.gz
tar xvf elpa-2017.11.001.tar.gz
cd elpa-2017.11.001
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh elpa
```

Please make the Intel Compiler and Intel MKL available on the command line. This depends on the environment. For instance, many HPC centers rely on `module load`.

```
source /opt/intel/compilers_and_libraries_2018.3.222/linux/bin/compilervars.sh intel64
```

For example, to configure and make for an Intel Xeon Scalable processor ("SKX"):

```
make clean
./configure-elpa-skx-omp.sh
make -j ; make install

make clean
./configure-elpa-skx.sh
make -j ; make install
```

ELPA Development To rely on experimental functionality, one may git-clone ELPA's master branch instead of downloading a regular version.

```
git clone --branch ELPA_KNL https://gitlab.mpcdf.mpg.de/elpa/elpa.git
```

To build ELPA, the instructions for building the latest release should apply as well.

References https://software.intel.com/en-us/articles/quantum-espresso-for-the-intel-xeon-phi-processor

LIBINT

Version 1.x For CP2K, LIBINT 1.1.x is required (1.2.x or 2.x cannot be used). Download and unpack LIBINT and make the configure wrapper scripts available in LIBINT's root folder. Please note that the "automake" package is a prerequisite.

```
wget --no-check-certificate https://github.com/evaleev/libint/archive/release-1-1-6.tar.gz
tar xvf release-1-1-6.tar.gz
cd libint-release-1-1-6
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh libint
```

Please make the Intel Compiler available on the command line. This depends on the environment. For instance, many HPC centers rely on `module load`.

```
source /opt/intel/compilers_and_libraries_2017.6.256/linux/bin/compilervars.sh intel64
```

For example, to configure and make for an Intel Xeon E5v4 processor (formerly codenamed "Broadwell"):

```
make distclean
./configure-libint-hsw.sh
make -j; make install
```

The version 1.x line of LIBINT does not support to cross-compile for an architecture (a future version of the wrapper scripts may patch this ability into LIBINT 1.x). Therefore, one can rely on the Intel Software Development Emulator (Intel SDE) to compile LIBINT for targets, which cannot execute on the compile-host.

```
/software/intel/sde/sde -knl -- make
```

To speed-up compilation, "make" might be carried out in phases: after "printing the code" (c-files), the make execution continues with building the object-file where no SDE needed. The latter phase can be sped up by interrupting "make", and executing it without SDE. The root cause of the entire problem is that the driver printing the c-code is (needlessly) compiled using the architecture-flags that are not supported on the host.

Further, for different targets (instruction set extensions) or different versions of the Intel Compiler, the configure scripts support an additional argument ("default" is the default tagname):

```
./configure-libint-hsw.sh tagname
```

As shown above, an arbitrary "tagname" can be given (without editing the script). This might be used to build multiple variants of the LIBINT library.

LIBXC

To configure, build, and install LIBXC 2.x, 3.x, and 4.x, one may proceed as shown below. Please note that CP2K 5.1 (and earlier) is only compatible with LIBXC 3.0 (or earlier, see also How to compile the CP2K code). Post-5.1, only the latest major release of LIBXC (by the time of the CP2K-release) is supported (e.g., LIBXC 4.x).

```
wget --content-disposition http://www.tddft.org/programs/octopus/down.php?file=libxc/4.3.4/libxc-4.3.4
tar xvf libxc-4.3.4.tar.gz
cd libxc-4.3.4
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh libxc
```

Please make the Intel Compiler available on the command line. This depends on the environment. For instance, many HPC centers rely on `module load`.

```
source /opt/intel/compilers_and_libraries_2017.6.256/linux/bin/compilervars.sh intel64
```

For example, to configure and make for an Intel Xeon Scalable processor ("SKX"):

```
make distclean
./configure-libxc-skx.sh
make -j; make install
```

LIBXSMM

LIBXSMM is a library targeting Intel Architecture (x86) for small, dense or sparse matrix multiplications, and small convolutions. The build instructions can be found at https://github.com/hfp/libxsmm (PDF).

QE

Build Instructions Download, unpack Quantum Espresso and make the configure wrapper scripts available in QE's root folder. Please note that the configure wrapper scripts support QE 6.x (prior support for 5.x is dropped). Before building QE, one needs to complete the recipe for ELPA.

NOTE: the ELPA configuration must correspond to the desired QE configuration e.g., `configure-elpa-skx-omp.sh` and `configure-qe-skx-omp.sh` ("omp").

```
wget https://gitlab.com/QEF/q-e/-/archive/qe-6.4.1/q-e-qe-6.4.1.tar.bz2
tar xvf q-e-qe-6.4.1.tar.bz2
cd q-e-qe-6.4.1
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh qe
```

Please make the Intel Compiler available on the command line, which may vary with the computing environment. For instance, many HPC centers rely on `module load`.

```
source /opt/intel/compilers_and_libraries_2018.5.274/linux/bin/compilervars.sh intel64
```

For example, configure for an Intel Xeon Scalable Processor (applicable to CPUs previously codenamed "Skylake" and "Cascadelake" server), and build the desired application(s) e.g., "pw", "cp", or "all".

```
./configure-qe-skx-omp.sh
make pw -j
```

Building "all" (or `make` without target argument) requires repeating `make all` until no compilation error occurs. This is because of some incorrect build dependencies (build order issue which might have been introduced by the configure wrapper scripts). In case of starting over, one can run `make distclean`, reconfigure the application, and build it again. For different targets (instruction set extensions) or different versions of the Intel Compiler, the configure scripts support an additional argument ("default" is the default tagname):

```
./configure-qe-skx-omp.sh tagname
```

As shown above, an arbitrary "tagname" can be given (without editing the script). This might be used to build multiple variants of QE. Please note: this tagname also selects the corresponding ELPA library (or should match the tagname used to build ELPA). Make sure to save your current QE build before building an additional variant!

Run Instructions To run Quantum Espresso in an optimal fashion depends on the workload and on the "parallelization levels", which can be exploited by the workload in question. These parallelization levels apply to execution phases (or major algorithms) rather than staying in a hierarchical relationship (levels). It is recommended to read some of the primary references explaining these parallelization levels (a number of them can be found in the Internet including some presentation slides). Time to solution may *vary by factors* depending on whether these levels are orchestrated or not. To specify these levels, one uses command line arguments along with the QE executable(s):

- `-npool`: try to maximize the number of pools. The number depends on the workload e.g., if the number of k-points can be distributed among independent pools. Indeed, per trial-and-error it is rather quick to check if a workload fails to pass the initialization phase. One may use prime numbers: *2*, *3*, *5*, etc. (default is *1*). For example, when *npool=2* worked it might be worth trying *npool=4*. On the other hand, increasing the number pools duplicates the memory consumption accordingly (larger numbers are increasingly unlikely to work).
- `-ndiag`: this number determines the number of ranks per pool used for dense linear algebra operations (DGEMM and ZGEMM). For example, if *64* ranks are used in total per node and *npool=2*, then put *ndiag=32* (QE selects the next square number which is less-equal than the given number e.g., *ndiag=25* in the previous example).
- `-ntg`: specifies the number of tasks groups per pool being used for e.g., FFTs. One can start with `NTG=$((NUMNODES*NRANKS/(NPOOL*2)))`. If `NTG` becomes zero, `NTG=${NRANKS}` should be used (number of ranks per node). Please note the given formula is only a rule of thumb, and the number of task groups also depends on the number of ranks as the workload is scaled out.

To run QE, below command line can be a starting point ("numbers" are presented as Shell variables to better understand the inner mechanics). Important for hybrid builds (MPI and OpenMP together) are the given environment variables. The `KMP_AFFINITY` assumes Hyperthreading (SMT) is enabled (granularity=fine), and the "scatter" policy allows to easily run less than the maximum number of Hyperthreads per core. As a rule of thumb, OpenMP adds only little overhead (often not worth a pure MPI application) but allows to scale further out when compared to pure MPI builds.

```
mpirun -bootstrap ssh -genvall \
  -np $((NRANKS_PER_NODE*NUMNODES)) -perhost ${NRANKS} \
  -genv I_MPI_PIN_DOMAIN=auto -genv I_MPI_PIN_ORDER=bunch \
  -genv KMP_AFFINITY=compact,granularity=fine,1 \
  -genv OMP_NUM_THREADS=${NTHREADS_PER_RANK} \
  /path/to/pw.x \<command-line-arguments\>
```

Performance An info-script (`info.sh`) is available attempting to present a table (summary of all results), which is generated from log files (use `tee`, or rely on the output of the job scheduler). There are only certain file extensions supported (`.txt`, `.log`). If no file matches, then all files (independent of the file extension) are attempted to be parsed (which will go wrong eventually). For legacy reasons (run command is not part of the log, etc.), certain schemes for the filename are eventually parsed and translated as well.

```
./run-qe.sh | tee qe-asrf112-4x16x1.txt
ls -1 *.txt
qe-asrf112-2x32x1.txt
qe-asrf112-4x16x1.txt

./info.sh [-best] /path/to/logs-or-cwd
```

AUSURF112	Nodes	R/N	T/R	Cases/d	Seconds	NPOOL	NDIAG	NTG
qe-asrf112-2x32x1	2	32	2	533	162.35	2	25	32
qe-asrf112-4x16x1	4	16	4	714	121.82	2	25	32

Please note that the number of cases per day (Cases/d) are currently calculated with integer arithmetic and eventually lower than just rounding down (based on 86400 seconds per day). The number of seconds taken are end-to-end (wall time), i.e. total time to solution including any (sequential) phase (initialization, etc.). Performance is higher if the workload requires more iterations (some publications present a metric based on iteration time).

References https://software.intel.com/en-us/articles/quantum-espresso-for-the-intel-xeon-phi-processor
http://www.quantum-espresso.org/wp-content/uploads/Doc/user_guide/node18.html

TensorFlow™ with LIBXSMM

There is a recipe available for TensorFlow with LIBXSMM (PDF). However, the recipe also contains information about building TensorFlow with Intel MKL and MKL-DNN (see section about Performance Tuning).

TensorFlow Serving

For experimentation, there is a recipe available for TensorFlow Serving with LIBXSMM. Please note this recipe is likely outdated and not intended for production use.