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 directly or use the documentation and then follow the build recipe of the desired application or library.

Documentation

- **ReadtheDocs**: online documentation with full text search: CP2K, ELPA, LIBINT, LIBXC, LIBXSMM, QE, and TF.
- **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 describes building CP2K with several (optional) libraries, which may be beneficial in terms of functionality and performance.

- Intel Math Kernel Library (also per Linux' distro's package manager) acts as:
  - LAPACK/BLAS and ScaLAPACK library
  - FFTw library
- LIBXSMM (replaces LIBSMM)
- LIBINT (depends on CP2K version)
- LIBXC (version 4.x)
- ELPA (depends on CP2K version)

The ELPA library eventually improves the performance (must be currently enabled for each input file even if CP2K was built with ELPA). There is also the option to auto-tune additional routines in CP2K (integrate/collocate) and to collect the generated code into an archive referred as LIBGRID.

For high performance, LIBXSMM (see also https://libxsmm.readthedocs.io) has been incorporated since CP2K 3.0. When CP2K is built with LIBXSMM, CP2K's "libsmm" library is not used and hence libsmm does not need to be built and linked with CP2K.

Getting Started

There are no configuration wrapper scripts provided for CP2K since a configure-step is usually not required, and the application can be built right away. CP2K's `install_cp2k_toolchain.sh` (under `tools/toolchain`) is out of scope in this document (it builds the entire tool chain from source including the compiler).

Although there are no configuration wrapper scripts for CP2K, below command delivers, e.g., an info-script and a script for planning CP2K execution:

```
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 scripts can be also download manually:

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

Step-by-step Guide

This step-by-step guide aims to build an MPI/OpenMP-hybrid version of the official release of CP2K by 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.

There are step-by-step guides for the current release (v7.1) and the previous release (v6.1).

Current Release This step-by-step guide uses (**a**) GNU Fortran (version 8.3, 8.4, or 9.2 are recommended, 9.1 is not recommended), or (**b**) Intel Compiler (version 19.1 "2020"). In any case, Intel MKL (2018, 2019, 2020 recommended) and Intel MPI (2018, 2020 recommended) need to be sourced. The following components are used:

- Intel Math Kernel Library (also per Linux' distro's package manager) acts as:
  - LAPACK/BLAS and ScaLAPACK library
  - FFTw library
- LIBXSMM (replaces LIBSMM)
- LIBINT (2.x from CP2K.org!)
- LIBXC (version 4.x)
- ELPA (version 2019.11.001)

To install Intel Math Kernel Library and Intel MPI from a public repository depends on the Linux distribution's package manager (mixing and matching recommended Intel components is possible). For newer distributions, Intel MKL and Intel MPI libraries are likely part of the official repositories. Otherwise a suitable repository must be added to the package manager (not subject of this document).

```
source /opt/intel/compilers_and_libraries_2020.0.166/linux/mpi/intel64/bin/mpivars.sh
source /opt/intel/compilers_and_libraries_2020.0.166/linux/mkl/bin/mklvars.sh intel64
```

If Intel Compiler is used, the following (or similar) makes the compiler and all necessary libraries available.

```
source /opt/intel/compilers_and_libraries_2020.0.166/linux/bin/compilervars.sh intel64
```

Please note, the ARCH file (used later/below to build CP2K) attempts to find Intel MKL even if the `MKLROOT` environment variable is not present. The MPI library is implicitly known when using compiler wrapper scripts (no need for `I_MPI_ROOT`). Installing the proper software stack and drivers for an HPC fabric to be used by MPI is out of scope in this document. If below check fails (GNU GCC only), the MPI's bin-folder must be added to the path.

```
$ mpif90 --version
  GNU Fortran (GCC) 8.3.0
  Copyright (C) 2018 Free Software Foundation, Inc.
  This is free software; see the source for copying conditions.  There is NO
  warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1) The first step builds ELPA. Please rely on ELPA 2019.

```
cd $HOME
wget http://elpa.mpcdf.mpg.de/html/Releases/2019.11.001/elpa-2019.11.001.tar.gz
tar xvf elpa-2019.11.001.tar.gz

cd elpa-2019.11.001
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh elpa
```

a) GNU GCC

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

b) Intel Compiler

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

Build and install ELPA:

```
make -j
make install
make clean
```

2) The second step builds LIBINT (preconfigured for CP2K).

```
cd $HOME
curl -s https://api.github.com/repos/cp2k/libint-cp2k/releases/latest \
| grep "browser_download_url" | grep "lmax-6" \
| sed "s/..*:␣\"\(..*[^\"]\)\".*/url \1/"␣\
|␣curl␣-LOK-
tar␣xvf␣libint-v2.6.0-cp2k-lmax-6.tgz
```

NOTE: A rate limit applies to GitHub API requests of the same origin. If the download fails, it can be worth trying an authenticated request by using a GitHub account (`-u "user:password"`).

```
cd libint-v2.6.0-cp2k-lmax-6
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh libint
```

NOTE: There are spurious issues about specific target flags requiring a build-system able to execute compiled binaries. To avoid cross-compilation (not supported here), please rely on a build sytem that matches the target system.

a) GNU GCC

```
./configure-libint-skx-gnu.sh
```

b) Intel Compiler

```
./configure-libint-skx.sh
```

Build and install LIBINT:

```
make -j
make install
make distclean
```

3) The third step builds LIBXC.

```
cd $HOME
wget --content-disposition https://gitlab.com/libxc/libxc/-/archive/4.3.4/libxc-4.3.4.tar.bz2
tar xvf libxc-4.3.4.tar.bz2

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
```

NOTE: Please disregard messages during configuration suggesting `libtoolize --force`.

a) GNU GCC

```
./configure-libxc-skx-gnu.sh
```

b) Intel Compiler

```
./configure-libxc-skx.sh
```

Build and install LIBXC:

```
make -j
make install
make distclean
```

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

```
cd $HOME
wget --no-check-certificate https://github.com/hfp/libxsmm/archive/1.15.tar.gz
tar xvf 1.15.tar.gz
```

5) This last step builds the PSMP-variant of CP2K. Please re-download the ARCH-files from GitHub as mentioned below (avoid reusing older/outdated files). If Intel MKL is not found, the key `MKLROOT=/path/to/mkl` can be added to Make's command line. To select a different MPI implementation one can try, e.g., `MKL_MPIRTL=openmpi`.

```
cd $HOME
wget https://github.com/cp2k/cp2k/releases/download/v7.1.0/cp2k-7.1.tar.bz2
tar xvf cp2k-7.1.tar.bz2
```

NOTE: Do not download the package `v7.1.0.tar.gz` from https://github.com/cp2k/cp2k/releases which was automatically generated by GitHub (it misses the source code from Git-submodules).

```
cd cp2k-7.1
wget --no-check-certificate https://github.com/hfp/xconfigure/raw/master/configure-get.sh
chmod +x configure-get.sh
./configure-get.sh cp2k
```

a) GNU GCC

```
rm -rf exe lib obj
make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=3 MIC=0 GNU=1 \
  LIBINTROOT=$HOME/libint/gnu-skx \
  LIBXCROOT=$HOME/libxc/gnu-skx \
  ELPAROOT=$HOME/elpa/gnu-skx-omp -j
```

b) Intel Compiler

```
rm -rf exe lib obj
make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=3 MIC=0 \
  LIBINTROOT=$HOME/libint/intel-skx \
  LIBXCROOT=$HOME/libxc/intel-skx \
  ELPAROOT=$HOME/elpa/intel-skx-omp -j
```

If no LIBXSMMMROOT was given (auto detection), the initial output of the build looks like:

```
Discovering programs ...
================================================================================
Automatically enabled LIBXSMM (LIBXSMMROOT=/path/to/libxsmm)
================================================================================
LIBXSMM release-1.15 (Linux)
--------------------------------------------------------------------------------
```

Once the build completed, the CP2K executable should be ready (`exe/Linux-x86-64-intelx/cp2k.psmp`):

```
$ LIBXSMM_VERBOSE=1 exe/Linux-x86-64-intelx/cp2k.psmp
  [...]
  LIBXSMM_VERSION: release-1.15
  LIBXSMM_TARGET: clx
```

Have a look at Running CP2K to learn more about pinning MPI processes (and OpenMP threads), and to try a first workload.

Previous Release As the step-by-step guide uses GNU Fortran (version 8.3 is recommended), only Intel MKL (2019.x recommended) and Intel MPI (2018.x recommended) need to be sourced (sourcing all Intel development tools of course does not harm). The following components are used:

- Intel Math Kernel Library (also per Linux' distro's package manager) acts as:
  - LAPACK/BLAS and ScaLAPACK library
  - FFTw library
- LIBXSMM (replaces LIBSMM)
- LIBINT (version 1.1.5 or 1.1.6)
- LIBXC (version 4.x)
- ELPA (version 2017.11.001)

NOTE: GNU GCC version 7.x or 8.x is highly recommended (CP2K built with GCC 9.1 or 9.2 may not pass regression tests).

```
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
```

To install Intel Math Kernel Library and Intel MPI from a public repository depends on the Linux distribution's package manager. For newer distributions, both libraries are likely part of the official repositories. Otherwise a suitable repository must be added to the package manager (not subject of this document). For example, installing with `yum` looks like:

```
sudo yum install intel-mkl-2019.4-070.x86_64
sudo yum install intel-mpi-2018.3-051.x86_64
```

Please note, the ARCH file (used later/below to build CP2K) attempts to find Intel MKL even if the `MKLROOT` environment variable is not present. The MPI library is implicitly known when using compiler wrapper scripts (no need for `I_MPI_ROOT`). Installing the proper software stack and drivers for an HPC fabric to be used by MPI is out of scope in this document. If below check fails, the MPI's bin-folder must be added to the path.

```
$ mpif90 --version
  GNU Fortran (GCC) 8.3.0
  Copyright (C) 2018 Free Software Foundation, Inc.
  This is free software; see the source for copying conditions.  There is NO
  warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

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 (1.1.6 recommended, newer version cannot be used). This library does not compile on an architecture with less CPU-features than the target (e.g., `configure-libint-skx-gnu.sh` implies to build on "Skylake" or "Cascadelake" server).

```
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 https://gitlab.com/libxc/libxc/-/archive/4.3.4/libxc-4.3.4.tar.bz2
tar xvf libxc-4.3.4.tar.bz2

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 --no-check-certificate https://github.com/hfp/libxsmm/archive/1.15.tar.gz
tar xvf 1.15.tar.gz
```

This last step builds the PSMP-variant of CP2K. Please re-download the ARCH-files from GitHub as mentioned below (avoid reusing older/outdated files). If Intel MKL is not found, the key `MKLROOT=/path/to/mkl` can be added to Make's command line. To select a different MPI implementation one can try, e.g., `MKL_MPIRTL=openmpi` (experimental: `patch -p0 src/mpiwrap/message_passing.F mpi-wrapper.diff`).

```
cd $HOME
wget https://github.com/cp2k/cp2k/archive/v6.1.0.tar.gz
tar xvf v6.1.0.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 AVX=3 MIC=0 GNU=1 \
  LIBINTROOT=$HOME/libint/gnu-skx \
  LIBXCROOT=$HOME/libxc/gnu-skx \
  ELPAROOT=$HOME/elpa/gnu-skx-omp -j
```

If no LIBXSMMMROOT was given (auto detection), the initial output of the build looks like:

```
Discovering programs ...
================================================================================
Automatically enabled LIBXSMM (LIBXSMMROOT=/path/to/libxsmm)
================================================================================
LIBXSMM release-1.15 (Linux)
--------------------------------------------------------------------------------
```

Once the build completed, the CP2K executable should be ready (`exe/Linux-x86-64-intelx/cp2k.psmp`):

```
$ LIBXSMM_VERBOSE=1 exe/Linux-x86-64-intelx/cp2k.psmp
  [...]
  LIBXSMM_VERSION: release-1.15
  LIBXSMM_TARGET: clx
```

Have a look at Running CP2K to learn more about pinning MPI processes (and OpenMP threads), and to try a first workload.

Intel Compiler

Below are the releases of the Intel Compiler, which are known to reproduce correct results according to the regression tests. It is also possible to mix and match different component versions by sourcing from different Intel suites.

- 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 and 2020: only suitable for CP2K 7.1 (and later)
  - source /opt/intel/compilers_and_libraries_2020.0.166/linux/bin/compilervars.sh intel64
  - Avoid 2019u1, 2019u2, 2019u3

- Intel MPI; usually any version is fine: Intel MPI 2018 and 2020 are recommended

NOTE: Intel Compiler 2019 (and likely later) is not recommended for CP2K 6.1 (and earlier).

Intel ARCH File

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 enough 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++) were key for passing CP2K's regression tests. If an own ARCH file is used or prepared, all libraries including LIBXSMM need to be built separately and referred in the link-line of the ARCH-file. In addition, CP2K may need to be informed and certain preprocessor symbols need to be given during compilation (`-D` compile flag). For further information, please follow the official guide and consider the CP2K Forum in case of trouble.

The purpose of the Intel ARCH files is to avoid writing an own ARCH-file even when GNU Compiler is used. Taking the Intel 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. The Intel ARCH files not only work with CP2K/Intel fork but even if an official release of CP2K is built (which is also encouraged). Of course, one can download the afore mentioned Intel 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
```

Running CP2K

Running CP2K may go beyond a single node, and pinning processes and threads becomes even more important. There are several schemes 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, in most cases CP2K prefers a total rank-count to be a square-number which leads to some complexity when aiming for rank/thread combinations that exhibit good performance properties. Please refer to the documentation of the script for planning MPI/OpenMP-hybrid (`plan.sh`), which illustrates running CP2K's PSMP-binary on an HT-enabled dual-socket system with 24 cores per processor/socket (96 hardware threads). The single-node execution with 16 ranks and 6 threads per rank looks like (`1x16x6`):

```
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
```

For an MPI command line targeting 8 nodes, `plan.sh` was used to setup 8 ranks per node with 12 threads per rank (`8x8x12`):

```
mpirun -perhost 8 -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=12 -genv I_MPI_DEBUG=4 \
  exe/Linux-x86-64-intelx/cp2k.psmp workload.inp
```

NOTE: the documentation of `plan.sh` also motivates and explains the MPI environment variables as shown in above MPI command lines.

Performance

The script for planning MPI-execution (`plan.sh`) is highly recommend along with reading the section about how to run CP2K. For CP2K, the MPI-communication patterns can be tuned in most MPI-implementations. For Intel MPI, the following setting can be beneficial:

```
export I_MPI_COLL_INTRANODE=pt2pt
export I_MPI_ADJUST_REDUCE=1
export I_MPI_ADJUST_BCAST=1
```

For large-scale runs, the startup can be tuned, but typically this is not necessary. However, the following may be useful (and does not harm):

```
export I_MPI_DYNAMIC_CONNECTION=1
```

Intel MPI usually nicely determines the fabric settings for both Omnipath and InfiniBand, and no adjustment is needed. However, people often prefer explicit settings even if it does not differ from what is determined automatically. For example, InfiniBand with RDMA can be set explicitly by using `mpirun -rdma` which can be also achieved with environment variables:

```
echo "'mpirun -rdma' and/or environment variables for InfiniBand"
export I_MPI_FABRICS=shm:dapl
```

As soon as several experiments are finished, it becomes handy to summarize the log-output. For this 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 by 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 "Cases/d" metric is calculated with integer arithmetic and hence represents fully completed cases per day (based on 86400 seconds per day). The number of seconds (as shown) is 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).

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.

Development

The Intel fork of CP2K was formerly a branch of CP2K's Git-mirror. CP2K is meanwhile natively hosted at GitHub. Ongoing work in the Intel branch was supposed to tightly track the master version of CP2K, which is also true for the fork. In addition, valuable topics may be upstreamed in a timelier fashion. To build CP2K/Intel from source for experimental purpose, 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 by using the `-xHost` target flag. For example, to explicitly target "Cascadelake" or "Skylake" server ("SKX"):

```
git clone https://github.com/hfp/libxsmm.git
git clone https://github.com/hfp/cp2k.git
cd cp2k
git submodule update --init --recursive

rm -rf lib obj
make ARCH=Linux-x86-64-intelx VERSION=psmp AVX=3 MIC=0
```

NOTE: 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`).

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 (like `ARCH=Linux-x86-64-intelx` or `VERSION=psmp`) can be passed at Make's command line when relying on CP2K/Intel's ARCH files.

- **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.

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>`).

References

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

ELPA

Build Instructions

ELPA 2019 Download and unpack ELPA and make the configure wrapper scripts available in ELPA's root folder.

NOTE: Please use ELPA 2017.11.001 for CP2K 6.1.

```
wget --no-check-certificate http://elpa.mpcdf.mpg.de/html/Releases/2019.11.001/elpa-2019.11.001.tar.gz
tar xvf elpa-2019.11.001.tar.gz
cd elpa-2019.11.001
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
```

Even if ELPA was just unpacked (and never built before), `make clean` is recommended in advance of building ELPA ("unknown module file format"). After building and installing the desired configuration(s), one may have a look at the installation:

```
[user@system elpa-2019.11.001]$ ls ../elpa
 intel-skx
 intel-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 2018 Please use ELPA 2017.11.001 for CP2K 6.1. For CP2K 7.1, please rely on ELPA 2019. ELPA 2018 **fails or crashes in several regression tests** in CP2K (certain rank-counts produce an incorrect decomposition), and hence ELPA 2018 should be avoided in production.

ELPA 2017 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: In Quantum Espresso, the `__ELPA_2018` interface must be used for ELPA 2017.11 (`-D__ELPA_2018`). The `__ELPA_2017` preprocessor definition triggers the ELPA1 legacy interface (get_elpa_row_col_comms, etc.), which was removed in ELPA 2017.11. This is already considered when using XCONFIGURE's wrapper scripts.

```
wget --no-check-certificate https://elpa.mpcdf.mpg.de/html/Releases/2017.11.001/elpa-2017.11.001.tar.g
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
```

Even if ELPA was just unpacked (and never built before), `make clean` is recommended in advance of building ELPA ("unknown module file format").

References

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

GROMACS

Build Instructions

Download, unpack GROMACS 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 server" 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

LIBINT

Overview

For CP2K 6.1 (and earlier), LIBINT 1.1.x is required (1.2.x, 2.x, or any later version cannot be used). For CP2K 7.x and onwards, LIBINT 2.5 (or later) is needed.

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_2018.5.274/linux/bin/compilervars.sh intel64
```

NOTE: CP2K 6.1 (and earlier) depend on LIBINT 1.1.x and a newer version of LIBINT cannot be used! CP2K 7.x (and later) rely on LIBINT 2.5 (or later) and cannot use the preconfigured library as provided on LIBINT's home page.

Version 2.5 (and later)

LIBINT generates code according to a configuration and an extent that is often specific to the application. The preconfigured downloads from LIBINT's home page may not be sufficient for CP2K and hence cannot be used. Please download (take "lmax-6" if unsure), unpack LIBINT, and make the configure wrapper scripts available in LIBINT's root folder.

To just determine the download-URL of the latest version (a suitable variant can be "lmax-6"):

```
curl -s https://api.github.com/repos/cp2k/libint-cp2k/releases/latest \
| grep "browser_download_url" | grep "lmax-6" \
| sed "s/..*:␣\"\(..*[^\"]\)\".*/\1/"
```

To download the lmax6-version right away, run the following command:

```
curl -s https://api.github.com/repos/cp2k/libint-cp2k/releases/latest \
| grep "browser_download_url" | grep "lmax-6" \
| sed "s/..*:␣\"\(..*[^\"]\)\".*/url \1/"␣\
|␣curl␣-LOK-
```

NOTE: A rate limit applies to GitHub API requests of the same origin. If the download fails, it can be worth trying an authenticated request by using a GitHub account (`-u "user:password"`).

To unpack the archive and to download the configure wrapper (lmax6-version is assumed):

```
tar xvf libint-v2.6.0-cp2k-lmax-6.tgz
cd libint-v2.6.0-cp2k-lmax-6

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

There are spurious issues about specific target flags requiring a build-system able to execute compiled binaries. To avoid cross-compilation (not supported here), please rely on a build sytem that matches the target system. For example, to configure and make for an Intel Xeon Scalable processor such as "Cascadelake" or "Skylake" server ("SKX"):

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

Make sure to run `make distclean` before reconfiguring a different variant, e.g., GNU and Intel variant. Further, for different targets (instruction set extensions) or different compilers, the configure-wrapper 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.

Version 1.x

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
```

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. If cross-compilation is necessary, 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.

LIBXC

To download, configure, build, and install LIBXC 2.x, 3.x, or 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/libxc/down.php?file=4.3.4/libxc-4.3.4.tar.gz
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_2018.5.274/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
```

NOTE: Please disregard messages during configuration suggesting `libtoolize --force`.

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-6.5/q-e-qe-6.5.tar.gz
tar xvf q-e-qe-6.5.tar.gz
cd q-e-qe-6.5
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_2020.0.166/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
https://www.quantum-espresso.org/Doc/user_guide/node1.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.

Appendix

CP2K MPI/OpenMP-hybrid Execution (PSMP)

Overview

CP2K's grid-based calculation as well as DBCSR's block sparse matrix multiplication (Cannon algorithm) prefer a square-number for the total rank-count (2d communication pattern). This is not to be obfuscated with a Power-of-Two (POT) rank-count that usually leads to trivial work distribution (MPI).

It can be more efficient to leave CPU-cores unused to achieve this square-number property than using all cores with a "wrong" total rank-count (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. Pinning MPI processes and placing threads requires extra care to be taken on a per-node basis to load a dual-socket system in a balanced fashion or to setup space between ranks for the OpenMP threads.

Because of the above-mentioned complexity, a script for planning MPI/OpenMP-hybrid execution (`plan.sh`) is available. Here is a first example for running the PSMP-binary on an SMP-enabled (Hyperthreads) dual-socket system with 24 cores per processor/socket (96 hardware threads in total). At first, a run with 48 ranks and 2 threads per core comes to the mind (`48x2`). However, for instance 16 ranks with 6 threads per rank may be better for performance (`16x6`). To easily place the ranks, Intel MPI is used:

```
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
```

NOTE: For hybrid codes, `I_MPI_PIN_DOMAIN=auto` is recommended as it spaces the ranks according to the number of OpenMP threads (`OMP_NUM_THREADS`). It is not necessary and not recommended to build a rather complicated `I_MPI_PIN_PROCESSOR_LIST` for hybrid codes (MPI plus OpenMP). To display and to log the pinning and thread affinization at the startup of an application, `I_MPI_DEBUG=4` can be used with no performance penalty. The recommended `I_MPI_PIN_ORDER=bunch` ensures that ranks per node are split as even as possible with respect to sockets (e.g., 36 ranks on a 2x20-core system are put in 2x18 ranks instead of 20+16 ranks).

Plan Script

To configure the plan-script, the metric of the compute nodes can be given for future invocations so that only the node-count is required as an argument. The script's help output (`-h` or `--help`) initially shows the "system metric" of the computer the script is invoked on. For a system with 48 cores (two sockets, SMP/HT enabled), setting up the "system metric" looks like (`plan.sh <num-nodes> <ncores-per-node> <nthreads-per-core> <nsockets-per-node>`):

```
./plan.sh 1 48 2 2
```

The script is storing the arguments (except for the node-count) as default values for the next plan (file: `$HOME/.xconfigure-cp2k-plan`). This allows to supply the system-type once, and to plan with varying node-counts in a convenient fashion. Planning for 8 nodes of the above kind yields the following output (`plan.sh 8`):

```
================================================================================
384 cores: 8 node(s) with 2x24 core(s) per node and 2 thread(s) per core
================================================================================
[48x2]: 48 ranks per node with 2 thread(s) per rank (14% penalty)
[24x4]: 24 ranks per node with 4 thread(s) per rank (14% penalty)
[12x8]: 12 ranks per node with 8 thread(s) per rank (33% penalty)
--------------------------------------------------------------------------------
[32x3]: 32 ranks per node with 3 thread(s) per rank (34% penalty) -> 16x16
[18x5]: 18 ranks per node with 5 thread(s) per rank (25% penalty) -> 12x12
[8x12]: 8 ranks per node with 12 thread(s) per rank (0% penalty) -> 8x8
[2x48]: 2 ranks per node with 48 thread(s) per rank (0% penalty) -> 4x4
--------------------------------------------------------------------------------
```

The first group of the output displays POT-style (trivial) MPI/OpenMP configurations (penalty denotes potential communication overhead), however the second group (if present) shows rank/thread combinations with the total rank-count hitting a square number (penalty denotes waste of compute due to not filling each node). For the given example, 8 ranks per node with 12 threads per rank may be chosen (`8x12`) and MPI-executed:

```
mpirun -perhost 8 -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=12 -genv I_MPI_DEBUG=4 \
  exe/Linux-x86-64-intelx/cp2k.psmp workload.inp
```

The script also suggests close-by configurations (lower and higher node-counts) that hit the square-property ("Try also the following node counts"). The example (as exercised above) was to illustrate how the script works, however it can be very helpful when running jobs especially on CPUs with not many prime factors in the core-count. Remember, the latter can be also the case for virtualized environments that reserve some of the cores to run the Hypervisor i.e., reporting less cores to the Operating System (guest OS) when compared to the physical core-count.

References

https://github.com/hfp/xconfigure/raw/master/config/cp2k/plan.sh
https://xconfigure.readthedocs.io/cp2k/