Dalton/LSDalton Installation Guide

Release 2016.0

Dalton/LSDalton developers

October 13, 2016

Contents

CHAPTER 1

Supported platforms and compilers

Many tests fail using Intel 2015. Therefore we currently do not recommend to compile Dalton 2016.0 using Intel 2015.

CHAPTER 2

Basic installation

2.1 General

Dalton is configured using CMake, typically via the `setup` script, and subsequently compiled using make or gmake. The `setup` script is a useful front-end to CMake. You need python to run `setup`.

2.2 Basics

To see all options, run:

```
$ ./setup --help
```

The `setup` script does nothing else than creating the directory "build" and calling CMake with appropriate environment variables and flags. In a typical installation we first configure with `setup` and then compile with `make`:

```
$ ./setup [--flags]
$ cd build
$ make
```

By default CMake builds out of source. This means that all object files and the final binary are generated outside of the source directory. Typically the build directory is called "build", but you can change the name of the build directory (e.g. "build_gfortran"):

```
$ ./setup [--flags] build_gfortran
$ cd build_gfortran
$ make
```

You can compile the code using all available several cores:

```
$ make -j
```

2.3 Typical examples

In order to get familiar with the configuration setup, let us demonstrate some typical configuration scenarios.

Configure for parallel compilation using MPI (make sure to properly export MPI paths):

```
$ ./setup --fc=mpif90 --cc=mpicc --cxx=mpicxx
```

There is a shortcut for it:

```
$ ./setup --mpi
```

Configure for parallel compilation using Intel MPI:

```
$ ./setup --fc=mpiifort --cc=mpiicc --cxx=mpiicpc
```

Configure for sequential compilation using ifort/icc/icpc and link against parallel mkl:

```
$ ./setup --fc=ifort --cc=icc --cxx=icpc --mkl=parallel
```

Configure for sequential compilation using gfortran/gcc/g++:

```
$ ./setup --fc=gfortran --cc=gcc --cxx=g++
```

Parallel compilation on a Cray:

```
./setup --fc=ftn --cc=cc --cxx=CC --cray --mpi
```

Parallel compilation on a SGI using Intel compilers and MPT:

```
./setup --fc=ifort --cc=icc --cxx=icpc --sgi-mpt
```

You get the idea. The configuration is usually good at detecting math libraries automatically, provided you export the proper environment variable `MATH_ROOT`, see *Linking to math libraries*.

2.4 What to do if CMake is not available or too old?

If it is your machine and you have an Ubuntu or Debian-based distribution:

```
$ sudo apt-get install cmake
```

On Fedora:

```
$ sudo yum install cmake
```

Similar mechanisms exist for other distributions or operating systems. Please consult Google.

If it is a cluster, please ask the Administrator to install/upgrade CMake.

If it is a cluster, but you prefer to install it yourself (it is easy):

1. Download the latest pre-compiled tarball from http://www.cmake.org/download/
2. Extract the downloaded tarball
3. Set correct PATH variable

CHAPTER 3

Installation instructions for system administrators

Please read the other installation sections for details but the installation procedure is basically this:

```
$ ./setup [--flags] --prefix=/full/install/path/
$ cd build
$ make [-jN]
$ ctest [-jN]
$ make install
```

This will install binaries, run scripts, the basis set library, as well as tools into the install path.

Advise users to always set a suitable scratch directory:

```
$ export DALTON_TMPDIR=/full/path/scratch
```

CHAPTER 4

Linking to math libraries

4.1 General

Dalton requires BLAS and LAPACK libraries. Typically you will want to link to external math (BLAS and LAPACK) libraries, for instance provided by MKL or Atlas.

By default the CMake configuration script will automatically detect these libraries:

```
$ ./setup --blas=auto --lapack=auto          # this is the default
```

if you define MATH_ROOT, for instance:

```
$ export MATH_ROOT='/opt/intel/mkl'
```

Do not use full path MATH_ROOT='/opt/intel/mkl/lib/ia32'. CMake will append the correct paths depending on the processor and the default integer type. If the MKL libraries that you want to use reside in /opt/intel/mkl/10.0.3.020/lib/em64t, then MATH_ROOT is defined as:

```
$ export MATH_ROOT='/opt/intel/mkl/10.0.3.020'
```

Then:

```
$ ./setup [--flags]
$ cd build
$ make
```

The math library detection will attempt to locate libraries based on MATH_ROOT, BLAS_ROOT/LAPACK_ROOT, MKL_ROOT, and MKLROOT.

4.2 Intel/MKL

If you compile with Intel compilers and have the MKL library available, you should use the –mkl flag which will automatically link to the MKL libraries (in this case you do not have to set MATH_ROOT). You have to specify whether you want to use the sequential or parallel (threaded) MKL version. For a parallel Dalton runs you should probably link to the sequential MKL:

```
$ ./setup --fc=mpif90 --cc=mpicc --cxx=mpicxx --mkl=sequential
```

For a sequential compilation you may want to link to the parallel MKL:

```
$ ./setup --fc=ifort --cc=icc --cxx=icpc --mkl=parallel
```

The more general solution is to link to the parallel MKL and control the number of threads using MKL environment variables.

4.3 Cray

Cray typically provides own optimized BLAS/LAPACK wrappers. For this use the option –cray to disable automatic BLAS/LAPACK detection:

```
$ ./setup --fc=ftn --cc=cc --cxx=CC --cray
```

4.4 Explicitly specifying BLAS and LAPACK libraries

If automatic detection of math libraries fails for whatever reason, you can always call the libraries explicitly like here:

```
$ ./setup --blas=/usr/lib/libblas.so --lapack=/usr/lib/liblapack.so
```

Alternatively you can use the –explicit-libs option. But in this case you should disable BLAS/LAPACK detection:

```
$ ./setup --blas=none --lapack=none --explicit-libs="-L/usr/lib -lblas -llapack"
```

4.5 Builtin BLAS and LAPACK implementation

If no external BLAS and LAPACK libraries are available, you can use the builtin implementation. However note that these are not optimized and you will sacrifice performance. This should be the last resort if nothing else is available:

```
$ ./setup --blas=builtin --lapack=builtin
```

4.6 LSDalton using ScaLAPACK/Intel/MKL

If you compile with Intel compilers and have the MKL library available, you can chose to compile LSDalton using the ScaLAPACK library provided by Intel. In this case you should set the MATH_ROOT environment variable and use the –scalapack flag which will automatically link to the MKL libraries.

You should not use the –mkl flag for this setup:

```
$ ./setup --fc=mpif90 --cc=mpicc --cxx=mpicxx --scalapack
```

CHAPTER 5

Scratch directory

Dalton and LSDalton need a scratch directory to write temporary files. Ideally this should be a fast-access disk.

You should always specify an explicit scratch directory with:

```
$ export DALTON_TMPDIR=/full/path/scratch
```

or by using:

```
$ dalton   -t /full/path/scratch [other flags and options]
$ lsdalton -t /full/path/scratch [other flags and options]
```

Which overrides `DALTON_TMPDIR`. If `DALTON_TMPDIR` is neither set nor passed to the run scripts, Dalton and LSDalton will search /global/work/$USER /scratch/$USER /work /scratch /scr /temp /tmp as candidates for a scratch directory. However you should not let (LS)Dalton default to those.

Do not point `DALTON_TMPDIR` to your home directory! To prevent loss of data (LS)Dalton always appends a directory to `DALTON_TMPDIR`.

CHAPTER 6

Basis set directory

The basis set directory is copied to the build directory (and possibly install directory). The `dalton` and `lsdalton` scripts will automatically find them. You can define or append custom basis set directories by exporting BASDIR:

```
export BASDIR='/somepath:/otherpath'
```

CHAPTER 7

Testing the installation

It is very important that you verify that your Dalton installation correctly reproduces the reference test set before running any production calculations.

The test set driver is CTest which can be invoked with "make test" after building the code.

7.1 Environment variables for testing

Before testing with "make test" you should export the following environment variables:

```
$ export DALTON_TMPDIR=/scratch            # scratch space for Dalton and LSDalton (adapt path)
$ export DALTON_LAUNCHER="mpirun -np 4"    # launch tests using 4 processes (only needed for Dalton)
$ export LSDALTON_LAUNCHER="mpirun -np 4"  # launch tests using 4 processes (only needed for LSDalton
```

Note that if you set the DALTON_NUM_MPI_PROCS to something different from 1, the dalton script will assume you have compiled using MPI and run the mpirun command!

7.2 Running the test set

You can run the whole test set either using:

```
$ make test
```

or directly through CTest:

```
$ ctest
```

Both are equivalent ("make test" runs CTest) but running CTest directly makes it easier to run sequential tests on several cores:

```
$ ctest -j4
```

You can select the subset of tests by matching test names to a regular expression:

```
$ ctest -R dft
```

Alternatively you can select the tests with a label matching a regular expression:

```
$ ctest -L rsp
```

The following command will give you all available labels:

```
$ ctest --print-labels
```

CHAPTER 8

Expert options

8.1 Compiling in verbose mode

Sometimes you want to see the actual compiler flags and definitions when compiling the code:

```
$ make VERBOSE=1
```

8.2 How can I change optimization flags?

You can turn optimization off (debug mode) like this:

```
$ ./setup --type=debug [other flags]
$ cd build
$ make
```

You can edit compiler flags in cmake/compilers/{FortranFlags.cmake, CFlags.cmake, CXXFlags.cmake}.

Alternatively you can edit compiler flags through ccmake:

```
$ cd build
$ ccmake ..
```

CHAPTER 9

How you can contribute to this documentation

These pages are rendered using RST/Sphinx and served using Read the Docs. RST is a subset of Sphinx. Sphinx is RST with some extensions.

9.1 How to modify the webpages

The source code for this documentation is hosted on GitLab. You need a GitLab account to modify the sources.

With a GitLab account you have two possibilities to edit the sources:

- If you are member of the dalton group, you can push directly to https://gitlab.com/dalton/installguide/. Once you commit and push, a webhook updates the documentation on http://dalton-installation.readthedocs.org/. This typically takes less than a minute.
- You fork https://gitlab.com/dalton/installguide/ and submit your changes at some point via a merge request. This means that your changes are not immediately visible but become so after a team member reviews your changes with a mouse click thus integrating them to https://gitlab.com/dalton/installguide/.

Note that the entire documentation including the entire documentation source code is public. Do not publish sensitive information and harvestable email addresses.

9.2 How to locally test changes

You do not have to push to see and test your changes. You can test them locally. For this install python-sphinx and python-matplotlib. Then build the pages with:

```
$ make html
```

Then point your browser to _build/html/index.html. The style is not the same but the content is what you would see after the git push.