Jacek Dziedzic
J.Dziedzic@soton.ac.uk
2021.03.18, v1.3

Instructions for compiling and running ONETEP on Young

1. Setting up the environment

You will need to load the correct set of modules. The provided config file is guaranteed to work with the Intel compiler v19.1.0.166 (curiously, known as `compilers/intel/2020`) and Intel MPI 2019 Update 6. To do this, add the following lines verbatim to the end of your `~/.bashrc` file:

```
module purge
module load rcps-core                      >/dev/null 2>/dev/null
module load userscripts                    >/dev/null 2>/dev/null
module load gcc-libs                       >/dev/null 2>/dev/null
module load git                            >/dev/null 2>/dev/null
module load compilers/intel/2020           >/dev/null 2>/dev/null
module load mpi/intel/2019/update6/intel >/dev/null 2>/dev/null
```

Then log out and log back in again for the changes to take effect. The first line takes care to unload all the modules so that they do not interfere with the ones we're using. Subsequent lines load the required modules – some prerequisites, git, the Intel compiler and Intel MPI.

2. Compilation

Use the config file `conf.young`. Issue the command below while in the directory of your ONETEP installation:

```
make onetep ARCH=young
```

This should compile and link a working binary, which will be placed in the `bin` subdirectory. If you tend to do this more often you might like the following command more:

```
make onetep ARCH=young –j | utils/colorise
```

It compiles many files in parallel (so it's faster) and colorises output to make it more readable.

3. Testing

If you have just compiled a new binary for Young, it would be prudent to test it by running a suite of quality-checks ("QCs") bundled with ONETEP. These test a number of important functionalities by running short jobs and comparing results against known good values. To run the QC-test suite, copy the QC-test submission script `qcsubmit.young.2021` (provided in the directory `hpc_resources/Young` of your ONETEP installtion) to the `tests` directory of your ONETEP installation. Then submit it to the batch system by changing to the `tests` directory and issuing

```
qsub jobsubmit.young.2021
```

There is no need to edit this file, unless you are using a budget (allocation) that is not `Soton_allocation`. If so, adjust the line marked with [*]. The tests should complete within two hours of starting. Once the testing job starts, you should see, in the same directory, a file called

`QC_tests.onnnnnn.out`, where *`nnnnnn`* is the ID (number) of your job. This contains the log of how your job is running – diagnostic messages from the batch system and the submission script, and any output from `testcode` – the python script that actually runs the tests. If everything goes well, you should see subsequent test numbers (there are about 85 of them), followed by "Passed" or, occasionally "WARNING". The warnings can usually be ignored. If there are any errors, the test number will be followed by "FAILED". If this happens, go to the directory of this particular test and examine the `test.out.*`, `test.err.*` and (potentially) `*.error_message` files. They should give you an idea about what went wrong. Once the tests complete, you will see a file called **`%DONE`** in your `tests` directory. This does not necessarily mean that all tests completed successfully. You should examine the `QC_tests-onnnnn.out` file for a line like this:

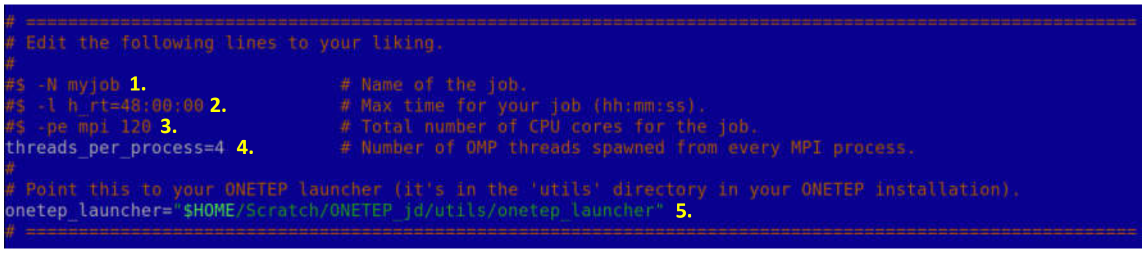
```
All done. 105 out of 105 tests passed (12 warnings).
```

If all tests passed – you're good to go.

4. Running

Use the provided submission script: `jobsubmit.young.2021`. Place it in a directory where the input files for your run are. Make sure you only run one calculation in a directory. Do not run multiple calculations (multiple `.dat` files) in a single directory – this will lead to a mess and the provided script disallows it.

Adjust the script to your liking by editing its first lines (see next page). Submit it with: `qsub jobsubmit.young.2021`. Details on how to monitor and control submitted and running jobs can be found on the Young support page @ https://www.rc.ucl.ac.uk/docs/Clusters/Young/



```
# ==========================================================================================
# Edit the following lines to your liking.
#
#$ -N myjob 1.                # Name of the job.
#$ -l h_rt=48:00:00 2.        # Max time for your job (hh:mm:ss).
#$ -pe mpi 120 3.             # Total number of CPU cores for the job.
threads_per_process=4  4.     # Number of OMP threads spawned from every MPI process.
#
# Point this to your ONETEP launcher (it's in the 'utils' directory in your ONETEP installation).
onetep_launcher="$HOME/Scratch/ONETEP_jd/utils/onetep_launcher" 5.
# ==========================================================================================
```

Things to adjust:

1. The name of your job. It has no effect on how it's run, but it will help you distinguish it from any other jobs you might have.
2. The walltime you are asking for in `hh:mm:ss`. The maximum on Young is `48:00:00`. Don't exceed this value or your job will never start. Normally using lower values allows faster job turnaround – the scheduler is able to stick short jobs into the allocation more easily.
3. The *total* number of **CPU cores** (MPI processes × OMP threads) you want to use, summed over all nodes. Each node on Young has **40 CPU cores**, so my example would use 3 full nodes, for a total of 120 CPU cores.
4. The number of OpenMP *threads* spawned from each MPI process. I suggest using 4. In this way each node can be saturated by 10 MPI processes spawning 4 OpenMP threads each. The parallel runner script on Young (`gerun`) will figure out the correct number of MPI tasks (processes), both per node and in total from the data provided in 3. and 4.
5. Path to the `onetep_launcher` script. Change it to the location of `onetep_launcher`. It is located in the `utils` subdirectory of the ONETEP installation. The script should be able

to find the correct ONETEP executable automatically (e.g. by examining `../bin`, relative to the `utils` directory it itself is in).

6. If you are using a budget (allocation) that is not `Soton_allocation`, adjust the line that says `Soton_allocation` to the correct allocation.

Once you adjusted the above, you are ready to submit your job. The submission script, together with `onetep_launcher`, will take of everything. There is no need for you to set the stack size, `OMP_STACKSIZE`, or `OMP_NUM_THREADS`. Also, **do not** add any `threads_` keywords to your input file. It's all handled by the script, really.

The submission script will try to communicate any error or success conditions by creating files with filenames beginning with `%` (so they are easy to spot in the file listing) in the directory of your run. These include:

`%NO_DAT_FILE` – there are no `.dat` files in your job directory. Job did not start.

`%MORE_THAN_ONE_DAT_FILE` – there are multiple `.dat` files in your job directory. Job did not start. This script only works correctly in a one-dat-file-per-run set-up.

`%ONETEP_LAUNCHER_MISSING` – the path you provided does not point to a valid `onetep_launcher`. It either does not exist or is not executable. Job did not start.

`%GERUN_ERROR` – `gerun`, the MPI wrapper script on Young reported an error. You job did not start or failed. Examine the standard error file for any error messages.

`%ONETEP_ERROR_DETECTED` – ONETEP failed gracefully, producing an error message. Job ran, but failed. Examine the `.error_message` file to see what happened.

`%ONETEP_DID_NOT_COMPLETE` – Job likely started, but it doesn't look like it's completed, even though there's no error message. It's likely your job deadlocked (got stuck), crashed non-gracefully or ran out of walltime. Examine the standard error file for any error messages and the ONETEP output for any hints on what might have happened.

`%DONE` – your calculation ran to completion with no apparent errors.

5. Crucial files.

Assuming your ONETEP input is called **`myinput.dat`**, these are some important files in your run.

- **`myinput.out`**: your ONETEP output.
- **`myinput.err`**: the standard error output of your job. On Young, once the job starts, it is immediately polluted by diagnostic information from `gerun`, such as the number of MPI processes (tasks) and OMP threads used, the names of the machines (nodes) on which your job is running and the `mpirun` command that `gerun` came up with to start your job. This will be all, if everything goes well. Otherwise, this file will also contain error messages from the OS, the batch system, `gerun`, MPI or the Fortran RTL. **You might wish to examine this file in case of any issues**.
- **`myinput.error_message`**: if ONETEP fails gracefully, this will contain an error message.

- ***jobname*.o*nnnnn*** and ***jobname*.po*nnnnn***, where ***jobname*** is the name of your job and ***nnnnn*** is the ID (number) of your job. This contains the log of how your job ran – diagnostic messages from the batch system and the submission script.
- ***jobname*.e*nnnnn*** and ***jobname*.pe*nnnnn***, where ***jobname*** is the name of your job and ***nnnnn*** is the ID (number) of your job. This contains any errors or messages of successful completion from the batch system and the submisstion script.
- **$modules_loaded**: The list of modules actually loaded when the job is run is echoed here. You might want to examine it if you have any doubts if they match your expectations.