Sustainable Packaging of Quantum Chemistry Software with the Nix Package Manager

Markus Kowalewski[*], Phillip Seeber[†]

October 13, 2021

Abstract

The installation of quantum chemistry software packages is commonly done manually and can be a time-consuming and complicated process. An update of the underlying Linux system requires a reinstallation in many cases and can quietly break software installed on the system. In this paper, we present an approach that allows for an easy installation of quantum chemistry software packages, which is also independent of operating system updates. The use of the Nix package manager allows building software in a reproducible manner, which allows for a reconstruction of the software for later reproduction of scientific results. The build recipes that are provided can be readily used by anyone to avoid complex installation procedures.

[*]markus.kowalewski@fysik.su.se, Department of Physics, Stockholm University, Sweden

[†]phillip.seeber@uni-jena.de, Institute of Physical Chemistry, Friedrich Schiller University Jena, Germany

1 Introduction

Open-source quantum chemistry program packages are usually compiled manually on a work station, single compute node, or a high-performance computing system. This process can be time-consuming and complex, especially when it has to be carried out for multiple programs. Such a manual installation is in general hard to replicate, unless its preparation and the use of configuration flags has been meticulously documented. A major problem with such an approach is that the program package will depend on operating system libraries, unless it has been linked completely statically. As a consequence, an update of the operating system or any other dependency may quietly break the software package and eventually make a rebuild necessary.

Another issue that arises from such a manual build strategy is the fact that scientific results can not be exactly reproduced. To rebuild an old version in exactly the same way, one needs also the exact states of all dependencies. This problem can in principle be solved by containerized solutions[1], such as docker[2] or singularity[3], but only works as long as the container image is preserved. Another downside of containerized solutions is that they usually ship with numerous system libraries which need to be updated or reproduced in case of an image rebuild.

The later reproduction of scientific results that were obtained via a computer program requires not only to follow the computational procedure, but also the exact same version of the program. This can only be guaranteed if one is able to reproduce the executable of the program. Package managers, which aim at creating reproducible build environments, are Nix[4] and Guix[5,6]. These package managers are built around functional languages,

which are used to create build recipes (derivations). A derivation becomes a functional expression with inputs (e.g., other derivations) and outputs (a path with the build product), which are tracked with cryptographic hashes. This approach to package management allows to uniquely identify a particular build of a program and provides all prerequisites to accurately reproduce the build at a later point in time. Exact binary reproducibility is difficult to achieve with traditional package managers, but can be achieved with the mechanisms provided by Nix.

In this paper, we show how the Nix package manager can be used to manage software in a sustainable and reproducible way. Our approach focuses mainly on quantum chemistry software packages, but can be applied to any software. We will introduce the *NixOS-QChem* overlay, which is an add-on to the *nixpkgs* collection for integrating quantum chemistry software into *nixpkgs* environment and providing optimized versions of the packages. The repository provides build recipes for open source and proprietary software packages.

The paper is organized as follows. In sec. 2 we give a general overview of the Nix package manager and its features. In sec. 3 we describe the approach to integrate quantum chemistry software packages along with a list of packaged software (sec. 3.1), followed by a set of examples in sec. 3.2.

2 Overview over the Nix package manager

The Nix package manager[4] is built around the Nix functional language[7] and a set of packages can be represented by a set of functions, which eventually

evaluate to a file system path. A build recipe is called a derivation in the Nix terminology. A derivation is a function, that can have one or more inputs and one or more outputs. The inputs are commonly other derivations, that provide dependencies, such as libraries. The output is a path to a final build product, such as the binaries of a package. The name of the output path is derived from a hash function over the derivation itself and all its inputs, which creates a unique path name.

The nix package manager stores its packages (i.e. its output paths) under a fixed path in the file system, `/nix/store`, which is simply called the nix store. Packages are not allowed to refer to dependencies outside the nix store, thus avoiding dependencies with the system software. All builds that are stored in the nix store are immutable and can not be changed after a build is completed, thus guaranteeing full stability of a given package.

The dependencies between nix store paths are tracked in a local database for proper handling by the Nix package manager. Every package (and every variation of it) is stored under a unique path name. As a result, many versions or variations of the same package can be installed in parallel in the nix store without interfering with the operating system's packages or interference between Nix store paths. Nix provides several mechanisms that allow for a user-friendly composition of the paths in the nix store into an environment for individual users. Only a selected set of nix store paths is projected into a user environment. Packages are either built on demand or downloaded from a binary cache, thus making the manual installation of a package unnecessary. No installation procedure by an administrator is thus necessary. This enables the end-user to use packaged software and create

their own builds.

Nix is, by design, a package manager which builds packages from source. If the output of a derivation is not available in the local nix store or in a remote binary cache, it will be built from source. This means that if a dependency of a package changes, the package will be rebuilt and potential problems with the update are either avoided or uncovered.

Note that the Nix approach differs substantially from the use of containers, which only statically bundle dependencies but provide no further mechanism to update, rebuild and maintain the contents of a container.

The second important component, besides Nix, is the *nixpkgs* package collection[8], which provides over 64,000 packages in the form of derivations[9] (including several quantum chemistry programs) and provides the basis for our work. The packages provided by the *nixpkgs* package collection are also available in form of binaries through a binary cache and thus do not need to be build from source by the end-user. This package set can be extended by the user with the help of overlays that allows us to add, modify, or replace packages. We will introduce *NixOS-QChem* overlay in sec 3, and show how it is used to add and optimize packages.

2.1 Nix in an HPC environment

In this section, we address the challenges that arise in a high performance computer cluster environment and how they can be addressed with Nix.

Environment modules[10] are an approach commonly used by super computing centers and on scientific computer clusters to create on-demand environments for users. A module sets environment variables pointing the

requested package paths upon a `module load <package>` call. However, a significant shortcoming of this approach is that it does not track dependencies between modules or any dependencies with system libraries. As a consequence, even a minor system update, addressing only security updates, may silently break installed packages or software compiled by a user.

The Nix package manager explicitly addresses these shortcomings. Users can choose to build their own packages with Nix or use a centrally provided package set. These user builds are independent of the operating system's software or centrally installed packages. The immutability of the nix store guarantees that a dependency on an existing package (nix store path) can never be quietly replaced. This allows users to pin a package and all its dependencies to a fixed version, providing stability and reproducibility of the binaries. When a dependent derivation is replaced or upgraded, it forces the rebuild of all derivations which depend on it, ensuring a valid build. The Nix package manager also provides an easy path for portability: packages can be transferred between different compute cluster systems (assuming that the nix package manager is installed on both systems) either as binary or by automatically rebuilding the required derivations from their source. Nix store paths and their dependencies can be transferred between machines by means of a custom copy command (`nix copy --[to|from] <machine> <store path>`).

Note that a proper installation of the nix package manager requires administrator rights and thus has to be carried out by a system administrator. The Gricad facility in Grenoble has demonstrated[11] how the nix package manager can be used on a computer cluster with a shared nix store. Nix has been used for example at CERN to distribute software for LHCb[12].

3 Structure of the Overlay and *nixpkgs*

To customise *nixpkgs* for use with quantum chemistry software packages, we make use of the overlay mechanism, which allows us to extend and modify the package set provided by *nixpkgs*. Note that many scientific libraries and some quantum chemistry packages are already packaged in *nixpkgs*. These packages can be used directly after the installation of the Nix package manager. Our overlay, *NixOS-QChem*[13], is thus tightly coupled to *nixpkgs*. The overlay serves multiple purposes: it selects quantum chemistry related software packages and adds additional quantum chemistry software packages that are not available in *nixpkgs*. The overlay also serves as an incubator for new packages that need to be matured first with respect to its integration into the *nixpkgs* environment. This includes packages that have non-standard build systems and are thus more difficult to integrate. The aim is to integrate useful variety of quantum chemistry packages into *nixpkgs* collection and to maintain a high code quality of the corresponding *nixpkgs* guidelines.

NixOS-QChem focuses on providing derivations for the x86-64 CPU architecture on the Linux platform, as this is currently the most common architecture for scientific high-performance computing[14]. The overlay also provides optional performance optimizations, which make use of modern x86-64 processors, that are not provided by *nixpkgs* itself due to compatibility reasons. The optimizations allow for setting custom compiler flags and automatically select optimization flags provided by individual packages.

All packages provided through the overlay are projected into a package subset (name prefix: `qchem`), which allows to also optimize basic libraries, such as the fftw library[15], without causing the rebuild of non-scientific soft-

ware packages. Open source packages can be downloaded automatically from the internet, but proprietary packages which require a license need to be provided by the user. For these cases, the overlay also provides a mechanism which allows for downloading from a custom, internal location. As a result, *NixOS-QChem* can provide derivations for commercial packages - such as Turbomole, Molpro and others - as well as packages that require user registration - such as CFour, MRCC, and ORCA[16–18]. While such packages often exclude the hurdles of compilation, packaging them enhances their composability.

Composing different major software packages in a single, coherent environment often proves difficult: the problems range from different providers of MPI and BLAS/LAPACK implementations for different codes and name conflicts in a global `$PATH` (e.g. `libblas.so`, `mpiexec`, ...), over different version constraints of the same dependencies (e.g. different version constraints of `numpy`[19] in different python packages, that cannot be fullfilled simultaneously). In those cases correct behaviour can become dependent on detailed choices, such as in which order different environment modules are loaded.

For selected packages, we have implemented automated tests in the overlay, which ensure that the basic functionality of a package is still given after an update or a rebuild. These tests are less comprehensive than the test suites provided by individual quantum chemistry packages, but aim at uncovering potential problems in connection with dependencies that have been observed during the integration.

The *nixpkgs* repository provides a simple mechanism to switch between libraries, either on a per-package basis or globally, for the whole package

set. One example is the message passing interface system (MPI)[20], which is provided by different implementations[21]. The default implementation is OpenMPI[22] but it can be readily replaced by the overlay mechanism. The following example shows the Nix code for an overlay that replaces OpenMPI globally with MVAPICH[23] and builds the CP2K[24] package explicitly with MPICH[25]:

```
self: super: {
  mpi = super.mvapich;
  cp2k = self.cp2k.override { mpi = super.mpich; };
}
```

Linear algebra libraries, such as BLAS and LAPACK can be replaced in a similar way. Nixpkgs has a wrapper for BLAS and LAPACK[26,27], which provides custom libraries through the standard interface. The default implementation is OpenBLAS[28], but Intel's MKL[29] or AMD's blis/libFlame[30] are also available. The following example demonstrates how an overlay can be used to replace BLAS and LAPACK with MKL:

```
self: super: {
  blas = super.blas.override {
    blasProvider = super.mkl;
  };
  lapack = super.lapack.override {
    lapackProvider = super.mkl;
  };
}
```

The Nix code in the *NixOS-QChem* overlay[13] is structured as follows:

- `default.nix`: the base of the overlay
- `cfg.nix`: defines all configuration options for the overlay
- `nixpkgs-opt.nix`: defines all packages from the *nixpkgs* collections

that are projected into the `qchem` subset and are subject to processor dependent optimisations.

- `tests/`: folder with tests for various packages.
- `examples/`: folder with examples showing different configuration scenarios.
- `pkgs/`: contains sub folders with derivations for additional packages.
- `install.sh`: installs nix, nixpkgs and the NixOS-QChem overlay.

3.1 List of Packaged Software

In combination, *nixpkgs* and *NixOS-QChem* provide a set of packages for quantum chemistry, molecular dynamics, and quantum dynamics (see table 1 for a subset of packages) that can be used directly in a production environment. The Nix derivations describe how to build software from source, and the builds are executed on demand. The package set is not restricted to free software and includes also derivations for proprietary packages. Many packages profit from this integrated packaging, and composing coherent runtime environments is simplified. Noteworthy examples for improved composability are the Pysisyphus optimiser[31], which wraps Turbomole, ORCA, and Psi4 among others, or the polarizable LICHEM QM/MM implementation, which relies on the Tinker MM engine and the Gaussian, NWChem, and Psi4 quantum chemistry codes. The SHARC surface hopping code, which depends on electronic structure codes, can be used conveniently from *NixOS-QChem*. SHARC requires deprecated Python2 as well as free and proprietary quantum

chemistry engines (BAGEL, Molcas, ORCA, Turbomole, Gaussian, Molpro). Using Nix an isolated environment with Python2 dependencies is available, quantum chemistry engines are directly provided to the SHARC scripts, and proper environment variables are set automatically, thus avoiding the error prone and difficult installation of multiple large quantum chemistry codes by the user.

To the best of the authors knowledge, the support for this variety of free and proprietary packages makes *NixOS-QChem* unique among such packaging efforts. With the DebiChem project of the Debian GNU/Linux distribution[32], another major packaging effort for chemical software exists. While the DebiChem team often provides valuable knowledge and patches, the architecture and philosophy of Debian packaging prevents clean isolation and tight integration between packages. Note that traditional package managers such as the Debian package manager are meant to be operated by system administrators and thus provide no straight forward way for end-user installations on a shared computer cluster.

3.2 Usage examples

We will outline the basic installation procedure of the Nix package manager and the overlay for a simple setup on a single machine. For the setup on a compute cluster, we refer to the setup of the Gricad team[11] for a shared nix store. A multi-user installation of Nix can be obtained with the following commands:

```
# To be executed by an admin
# Multi-user installation of Nix.
```

Table 1: List of selected quantum chemistry packages and utilities provided by the overlay.

Package	Attribute	Reference
avogadro2-1.94.0	qchem.avogadro2	[33]
bagel-1.2.2	qchem.bagel	[34,35]
cfour-2.1	qchem.cfour	[16]
cp2k-8.2.0	qchem.cp2k	[24]
crest-2.11.1	qchem.crest	[36]
dalton-2020.0	qchem.dalton	[37]
ergoscf-3.8	qchem.ergoscf	[38]
gpaw-21.6.0	qchem.gpaw	[39]
gromacs-2020.4	qchem.gromacs	[40]
nwchem-7.0.2	qchem.nwchem	[41]
molden-6.3	qchem.molden	[42]
molpro-2021.2.0	qchem.molpro	[17]
mrcc-2020.02.22	qchem.mrcc	[43]
octopus-10.3	qchem.octopus	[44]
openmolcas-21.06	qchem.molcas	[45]
orca-5.0.1	qchem.orca	[18]
pcmsolver-1.3.0	qchem.pcmsolver	[46]
psi4-1.4	qchem.psi4	[47]
pyscf-1.7.6.post1	qchem.python3.pkgs.pyscf	[48]
pysisyphus-0.7.2	qchem.pysisyphus	[31]
quantum-espresso-6.6	qchem.quantum-espresso	[49]
sharc-2.1.1	qchem.sharc	[50]
siesta-4.1-b3	qchem.siesta	[51]
tinker-8.8.3	qchem.tinker	[52]
turbomole-7.5.1	qchem.turbomole	[53]
vmd-1.9.3	qchem.vmd	[54]
xcfun-2.1.1	qchem.xcfun	[55]
xtb-6.4.1	qchem.xtb	[56]

```
# Will request root privileges for the inital setup.
sh $> curl -L https://nixos.org/nix/install |
        sh -s -- --daemon

# To be executed by a user
# Setup of a Nix package channel
sh $> nix-channel --add \
      https://nixos.org/channels/nixpkgs-unstable \
      nixpkgs
sh $> nix-channel --update

# Make user channels available
sh $> echo 'export␣NIX_PATH=nixpkgs=\
␣␣␣␣␣␣$HOME/.nix-defexpr/channels/nixpkgs\
␣␣␣␣␣␣:$HOME/.nix-defexpr/channels\
␣␣␣␣␣␣:/nix/var/nix/profiles/per-user/root/channels'\
      >> ~/.bashrc
sh $> source ~/.bashrc
```

These commands will install Nix in multi-user mode; a Nix daemon will listen for evaluation requests from the Nix commands and execute builds or download the store paths from a binary cache.

The packages in the *NixOS-QChem* can be accessed with different methods. We will discuss two main methods here: as a direct system-wide or user-installed overlay to the *nixpkgs* channel and explicit use as a project-based package source. The first method allows for a direct use of the latest package versions, while the second method allows to fix the version on a per-project basis. Other options to access *NixOS-QChem* overlay packages, which we will not discuss here further in detail, are the Nix User Repositories (NUR)[57], the experimental Nix flakes feature, or a customised Nix channel. None of the above variants is mutual exclusive and each one can be useful for different scenarios. NUR and the channel mechanism provide convenient automatic updates, while flakes provide hermetic expressions, that are not influenced by the runtime environment, and the implicit overlay yields a convenient compostion of the package set.

The *NixOS-QChem* overlay can be used as an implicit overlay by placing the repository in a directory reconginzed by *nixpkgs*:

```
sh $> mkdir -p ~/.config/nixpkgs/overlays
sh $> git clone \
      https://github.com/markuskowa/NixOS-QChem.git \
      ~/.config/nixpkgs/overlays/qchem
```

Packages from the overlay are then available for use via Nix commands, e.g. `nix-shell -p qchem.xtb`. Updates to the overlay happen explicitly by calling `git pull`. The behaviour of *nixpkgs* and *NixOS-QChem* can be controlled by settings in `~/.config/nixpkgs/config.nix`, which allows to enabl the build of proprietary packages and apply CPU related tuning options. The Nix code of a configuration that enables AVX2 performance tuning for CPUs from the Haswell generation onwards, and allows using proprietary packages, is given by the following example:

```
{
  config = {
    # Attempt build of packages
    # with non open source licences
    allowUnfree = true;
    qchem-config = {
      # Enable AVX2 CPU optmisations
      # (Haswell CPU target).
      optAVX = true;
      # Molpro license token if available
      licMolpro = null;
    };
  }
}
```

Nix can serve different use cases for computational tasks: installing a package in the user's environment, launching an isolated shell, interactive use of a program, or the noninteractive execution of programs in a resource manager like SLURM. To exemplify some common use cases, we will refer to

illustrative examples in the following.

Interactive Program Usage (Turbomole) : Turbomole uses a set of interactive programs, such as `define` and `eiger`, to create input files and analyse output files. Furthermore, Turbomole requires environment variables such as `$TURBODIR` and `$PARA_ARCH` to be set. An interactive `nix-shell` makes the Turbomole package available and reduces the required user input, by wrapping Turbomole with appropriate environment variables and settings:

```
# starts a interative nix-shell with Turbomole
sh $> nix-shell -p qchem.turbomole

# Turbomole commands can directly be used
# normal interaction with define
# e.g. set up a RI-ADC(2) calculation
nix-shell $> define
# ground state calculation
nix-shell $> ridft -smpcpus 4
# excited state calculation
nix-shell $> ricc2 -smpcpus 4
# interactive overview of results
nix-shell $> eiger
# will drop back to normal bash
nix-shell $> exit
```

Non-Interactive Calculation (Molcas) A noninteractive, Molcas calculation with OMP parallelism can directly be executed from a `nix-shell`. The PyMolcas driver requires specfic Python packages, such as `six`, to be installed. Instead of globally installing Python dependencies, the Nix derivation wraps the python scripts in an isolated python runtime environment and can be used directly:

```
sh $> nix-shell \
        -p qchem.molcas \
        --run "OMP_NUM_THREADS=4
                    pymolcas molcas.inp"
```

Interactive Python Session (MEEP) Some scientific Python packages may be used interactively within an interpreter, e.g. to experiment with different settings. Packages such as MEEP[58], that provide a Python API around a C/C++ code are often difficult to install; they are not available from PyPi and require both Python and C/C++ tooling. MEEP can be used interactively from Python within a `nix-shell`:

```
sh $> nix-shell \
          -p python3 python3.pkgs.numpy \
             qchem.python3.pkgs.meep \
          --run "python3"
python3 $> import numpy as np
python3 $> import meep as mp
python3 $> # ...
```

Project-Based Calculation Environment with Fixed Versions Computational environments, that are associated with a specific project, can strongly benefit from fixing all package versions in a custom environment. Projects can use different versions or variations of programs without interfering with a system level package set. Such a computational environment can in principle be defined in a single nix file and thus be easily shared between coworkers. Fixing all program versions in such an environment also allows to reproduce its results at a later point in time. Such an environment can be described by a `shell.nix` file, which defines an environment for a `nix-shell`. To achieve reproducibility, the versions of *nixpkgs* and *NixOS-QChem* must be fixed:

```
let
 # Reproducible, pinned import of the
 # NixOS-QChem overlay function
 gh = "https://github.com";
```

```
 qchemOvl = import (builtins.fetchGit {
  url = "${gh}/markuskowa/NixOS-QChem.git";
  name = "NixOS-QChem_2021-09-25";
  rev = "9604e9b7f8d6ea68f07d621e1f70a9ebf857efa0";
  ref = "master";
 });

 nixpkgs = import (builtins.fetchGit {
  url = "${gh}/NixOS/nixpkgs.git";
  name = "nixpkgs_2021-09-25";
  rev = "a3a23d9599b0a82e333ad91db2cdc479313ce154";
  ref = "nixpkgs-unstable";
 });

 pkgs = nixpkgs {
   overlays = [ qchemOvl ];
   config = { ... };
 };

in with pkgs; mkShell { ... }
```

Here, the `fetchGit` function is used to access a specific version of the overlay, and the *nixpkgs* package set (fixed by the respective `rev` statements). Alternatively, the Niv tool[59] provides a convenient command line interface to automate the version fixing and update processes. The overlay and configuration settings are applied explicitly in the `shell.nix` file. For the rather verbose, full example of the `shell.nix` file and the usage of Niv, we refer to[60]. The `shell.nix` file can either be referenced implicitly by executing `nix-shell` in the same directory, or explicitly by `nix-shell /path/to/shell.nix`.

Reproducible Jupyter Notebooks Jupyter notebooks[61] are commonly used tools for experimentation with codes and methods, the development of scientific ideas, as well as for visualization of data. However, distributing Jupyter notebooks can be difficult, since the environment and all dependencies, such as Python packages, also needs to be reproduced. Like in the previous example, `nix-shell` can be used to make Jupyter notebooks repro-

ducible. Using version fixing, as in the example above, a Jupyter environment for GPAW simulations can be formulated in a `shell.nix` file:

```
let
  qchemOvl = ...
  pkgs = ...
  pythonWithPackages =
    pkgs.qchem.python3.withPackages
    (p: with p; [
      numpy
      jupyterlab
      ipympl
      gpaw
    ]);
in with pkgs; mkShell {
  buildInputs = [ pythonWithPackages ];
  shellHook = "jupyter-lab";
}
```

Executing `nix-shell` will then directly open the Jupyter-Lab interface in the browser and allow using packages such as GPAW, along with Python and all the necessary Python packages. The complete examples can be found in Ref.[60].

Self-Contained Programs and Shell Scripts The `nix-shell` command can be used as the shebang line of scripts. This allows to write small, reproducible, self-contained scripts and programs, or to write scripts in the scope of a project-associated `shell.nix` file. The following example shows a self-contained Python script for data visualization:

```
#! /usr/bin/env nix-shell
#! nix-shell -i python3
#! nix-shell -p python3Packages.numpy
#! nix-shell -p python3Packages.matplotlib

import numpy as np
import matplotlib.pyplot as plt

xs = np.linspace(-2, 2, num=100)
plt.plot(xs, np.exp(-xs**2))
```

```
plt.show()
```

Note, that here we use the latest versions of numpy and matplotlib as they are provided directly by *nixpkgs*.

This mechanism can also be used to write SLURM (or other resource management system) scripts for working in a computer cluster environment. Such batch scripts can either pull in packages via `nix-shell`'s `-p` option or can be combined with a project-associated `shell.nix` file

```
#! /usr/bin/env nix-shell
#! nix-shell /path/to/project/shell.nix -i bash

#SBATCH --ntasks=360
#SBATCH --ntasks-per-node=36
#SBATCH --nodes=10
#SBATCH --mem=0
#SBATCH --partition=s_standard

mpiexec \
  -np $SLURM_NTASKS \
  --map-by ppr:$SLURM_TASKS_PER_NODE:node \
  nwchem input.nw > output.log
```

Reusing the environment from the project's `shell.nix` ensures to have exactly the same computational environment on the compute nodes, the front-end node, or the user's local work station. This eliminates error-prone `module load` operations, and ensures independence of potentially different system libraries between nodes. These scripts can also be easily transferred between Nix enabled computing centers.

4 Conclusion and Outlook

The *nixpkgs* set and the *NixOS-QChem* overlay provide numerous scientific packages and packages relevant for quantum chemistry. The presented solu-

tion makes these programs easily available without complicated installation or manual compilation procedures. Proprietary packages can also be made available without explicit installation if the user has obtained a license and has the corresponding installation file. The *NixOS-QChem* overlay is configurable and allows for optimization depending on the used processor architectures. The option to build self-contained scripts and batch jobs has proven itself highly useful in daily use. The presented examples demonstrate how to create reproducible environments for electronic structure calculations as for scripted pre- and post-processing tasks.

The presented approach is focused on applications for the theoretical chemistry community, but the general principle is of broad applicability. We think that many scientific applications would benefit from the Nix approach. Reproducible environments are not only useful for users of scientific software, but are also helpful during the development of software.

Future developments of the *NixOS-QChem* overlay will aim at integrating more quantum chemistry software packages with Nix. We encourage users and developers of scientific software to contribute to *NixOS-QChem* and *nixpkgs* as well as to report bugs. It would be a great advantage if more computing facilities will adopt the approach and provide a Nix installation to allow for more reproducible compute environments.

5 Acknowledgments

The authors would like to thank to the *nixpkgs* community for providing the software infrastructure, that made this work possible. Phillip Seeber grate-

fully acknowledges the financial support provided by the German Research Foundation within the TRR CATALIGHT – Projektnummer 364549901-TRR234 (project C5).

References

1. O. S. Navarro Leija, K. Shiptoski, R. G. Scott, B. Wang, N. Renner, R. R. Newton, and J. Devietti, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Association for Computing Machinery, New York, NY, USA, 2020), ASPLOS '20, p. 167–182, ISBN 9781450371025, URL `https://doi.org/10.1145/3373376.3378519`.

2. D. Merkel, Linux journal **2014**, 2 (2014).

3. G. M. Kurtzer, V. Sochat, and M. W. Bauer, PLos ONE **12**, e0177459 (2017).

4. E. Dolstra, Ph.D. thesis, Utrecht University (2006).

5. L. Courtès and R. Wurmus, in *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)* (Vienne, Austria, 2015), URL `https://hal.inria.fr/hal-01161771`.

6. *Guix hpc: Reproducible software deployment for high-performance computing.*, `https://hpc.guix.info/`, accessed: 2021-08-26.

7. *Nix manual*, `https://nixos.org/manual/nix/stable/`, accessed: 2021-09-13.

8. *Nixpkgs manual*, https://nixos.org/manual/nixpkgs/stable/, accessed: 2021-09-10.

9. *Repology, the packaging hub*, https://repology.org/repositories/statistics/total, accessed: 2021-08-31.

10. *Environment modules*, https://modules.readthedocs.io/en/latest/index.html, accessed: 2021-09-10.

11. B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard, in *Proceedings of the Fourth International Workshop on HPC User Support Tools* (Association for Computing Machinery, New York, NY, USA, 2017), HUST'17, ISBN 9781450351300, URL https://doi.org/10.1145/3152493.3152556.

12. Burr, Chris, Clemencic, Marco, and Couturier, Ben, EPJ Web Conf. **214**, 05005 (2019), URL https://doi.org/10.1051/epjconf/201921405005.

13. https://github.com/markuskowa/NixOS-QChem, accessed: 2021-09-13.

14. *Top 500 the list.*, https://www.top500.org/statistics/list/, accessed: 2021-09-21.

15. M. Frigo and S. Johnson, Proc. IEEE **93**, 216 (2005).

16. D. A. Matthews, L. Cheng, M. E. Harding, F. Lipparini, S. Stopkowicz, T.-C. Jagau, P. G. Szalay, J. Gauss, and J. F. Stanton, The Journal of Chemical Physics **152**, 214108 (2020), URL https://doi.org/10.1063%2F5.0004837.

17. H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, and M. Schütz, Wiley Interdisciplinary Reviews: Computational Molecular Science **2**, 242 (2012), URL https://doi.org/10.1002%2Fwcms.82.

18. F. Neese, Wiley Interdisciplinary Reviews: Computational Molecular Science **2**, 73 (2012).

19. C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., Nature **585**, 357 (2020), URL https://doi.org/10.1038/s41586-020-2649-2.

20. M. P. Forum, Tech. Rep., USA (1994).

21. *Nixpkgs manual: Switching the mpi implementation*, https://nixos.org/manual/nixpkgs/stable/#sec-overlays-alternatives-mpi, accessed: 2021-09-15.

22. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al., in *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, 2004), pp. 97–104.

23. D. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, Journal of Computational Science **52**, 101208 (2020).

24. T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, et al., J. Chem. Phys. **152**, 194103 (2020), https://doi.org/10.1063/5.0007045, URL https://doi.org/10.1063/5.0007045.

25. W. Gropp, in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Springer-Verlag, Berlin, Heidelberg, 2002), p. 7, ISBN 3540442960.

26. L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al., ACM Transactions on Mathematical Software **28**, 135 (2002).

27. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., *LAPACK Users' Guide* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999), 3rd ed., ISBN 0-89871-447-8 (paperback).

28. *Openblas: An optimized blas library*, `http://www.openblas.net/`, accessed: 2021-09-21.

29. *Intel oneapi math kernel library*, `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html#gs.bbhali`, accessed: 2021-09-21.

30. *Amd optimizing cpu libraries (aocl), blas library*, `https://developer.amd.com/amd-aocl/blas-library/`, accessed: 2021-09-28.

31. J. Steinmetzer, S. Kupfer, and S. Gräfe, International Journal of Quantum Chemistry **121** (2021), URL `https://doi.org/10.1002%2Fqua.26390`.

32. *Debichem is a debian pure blend targeted at chemistry*, `https://wiki.debian.org/Debichem`, accessed: 2021-09-21.

33. K. Sharkey, M. D. Hanwell, C. Harris, and A. Vacanti, The Source **26** (2013), https://blog.kitware.com/avogadro-2-and-open-chemistry/, URL https://blog.kitware.com/avogadro-2-and-open-chemistry/.

34. *Bagel, brilliantly advanced general electronic-structure library. http://www.nubakery.org under the gnu general public license.*, http://www.nubakery.org, accessed: 2021-09-10.

35. T. Shiozaki, WIREs Comput. Mol. Sci. **8**, e1331 (2018), https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.1331, URL https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1331.

36. P. Pracht, F. Bohle, and S. Grimme, Physical Chemistry Chemical Physics **22**, 7169 (2020), URL https://doi.org/10.1039%2Fc9cp06869d.

37. K. Aidas, C. Angeli, K. L. Bak, V. Bakken, R. Bast, L. Boman, O. Christiansen, R. Cimiraglia, S. Coriani, P. Dahle, et al., Wiley Interdiscip Rev Comput Mol Sci **4**, 269 (2014), URL https://www.ncbi.nlm.nih.gov/pubmed/25309629.

38. E. Rudberg, E. H. Rubensson, P. Sałek, and A. Kruchinina, SoftwareX **7**, 107 (2018).

39. J. Enkovaara, C. Rostgaard, J. J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. A. Hansen, et al., J. Phys.: Condens. Matter **22**, 253202 (2010).

40. M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, SoftwareX **1-2**, 19 (2015), URL https://doi.org/10.1016%2Fj.softx.2015.06.001.

41. E. Aprà, E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, H. J. J. van Dam, Y. Alexeev, J. Anchell, et al., J. Chem. Phys. **152**, 184102 (2020).

42. G. Schaftenaar and J. M. Noordik, J. Comput. Aided. Mol. Des. **14**, 123 (2000).

43. M. Kállay, P. R. Nagy, D. Mester, Z. Rolik, G. Samu, J. Csontos, J. Csóka, P. B. Szabó, L. Gyevi Nagy, B. Hégely, et al., J. Chem. Phys. **152**, 074107 (2020), URL https://doi.org/10.1063%2F1.5142048.

44. N. Tancogne-Dejean, M. J. T. Oliveira, X. Andrade, H. Appel, C. H. Borca, G. Le Breton, F. Buchholz, A. Castro, S. Corni, A. A. Correa, et al., J. Chem. Phys. **152**, 124119 (2020), https://doi.org/10.1063/1.5142502, URL https://doi.org/10.1063/1.5142502.

45. I. Fdez. Galván, M. Vacher, A. Alavi, C. Angeli, F. Aquilante, J. Autschbach, J. J. Bao, S. I. Bokarev, N. A. Bogdanov, R. K. Carlson, et al., J. Chem. Theo. Comput. **15**, 5925 (2019), pMID: 31509407, https://doi.org/10.1021/acs.jctc.9b00532, URL https://doi.org/10.1021/acs.jctc.9b00532.

46. R. Di Remigio, A. H. Steindal, K. Mozgawa, V. Weijo, H. Cao, and L. Frediani, International Journal of Quantum Chemistry **119**, e25685 (2019), https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.

25685, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/qua.25685.

47. D. G. A. Smith, L. A. Burns, A. C. Simmonett, R. M. Parrish, M. C. Schieber, R. Galvelis, P. Kraus, H. Kruse, R. Di Remigio, A. Alenaizan, et al., The Journal of Chemical Physics **152**, 184108 (2020), URL https://doi.org/10.1063%2F5.0006002.

48. Q. Sun, T. C. Berkelbach, N. S. Blunt, G. H. Booth, S. Guo, Z. Li, J. Liu, J. D. McClain, E. R. Sayfutyarova, S. Sharma, et al., WIREs Computational Molecular Science **8** (2018), URL https://doi.org/10.1002%2Fwcms.1340.

49. P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, et al., Journal of Physics: Condensed Matter **21**, 395502 (2009), URL https://doi.org/10.1088%2F0953-8984%2F21%2F39%2F395502.

50. S. Mai, M. Richter, M. Heindl, M. F. S. J. Menger, A. Atkins, M. Ruckenbauer, F. Plasser, L. M. Ibele, S. Kropf, M. Oppel, et al., *Sharc2.1: Surface hopping including arbitrary couplings — program package for non-adiabatic dynamics*, sharc-md.org (2019).

51. A. García, N. Papior, A. Akhtar, E. Artacho, V. Blum, E. Bosoni, P. Brandimarte, M. Brandbyge, J. I. Cerdá, F. Corsetti, et al., The Journal of Chemical Physics **152**, 204108 (2020), https://doi.org/10.1063/5.0005077, URL https://doi.org/10.1063/5.0005077.

52. J. A. Rackers, Z. Wang, C. Lu, M. L. Laury, L. Lagardère, M. J. Schnieders, J.-P. Piquemal, P. Ren, and J. W. Ponder, Journal of Chemical Theory and Computation **14**, 5273 (2018), URL https://doi.org/10.1021%2Facs.jctc.8b00529.

53. *TURBOMOLE V7.5.1 2021, a development of University of Karlsruhe and Forschungszentrum Karlsruhe GmbH, 1989-2007, TURBOMOLE GmbH, since 2007; available from* http://www.turbomole.com.

54. W. Humphrey, A. Dalke, and K. Schulten, Journal of Molecular Graphics **14**, 33 (1996).

55. U. Ekström, L. Visscher, R. Bast, A. J. Thorvaldsen, and K. Ruud, Journal of Chemical Theory and Computation **6**, 1971 (2010), URL https://doi.org/10.1021%2Fct100117s.

56. C. Bannwarth, E. Caldeweyher, S. Ehlert, A. Hansen, P. Pracht, J. Seibert, S. Spicher, and S. Grimme, WIREs Computational Molecular Science **11** (2021), URL https://doi.org/10.1002%2Fwcms.1493.

57. *Nur - nix user repository: User contributed nix packages*, https://github.com/nmattia/niv, accessed: 2021-09-24.

58. A. F. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. Joannopoulos, and S. G. Johnson, Computer Physics Communications **181**, 687 (2010), URL https://doi.org/10.1016%2Fj.cpc.2009.11.008.

59. *Niv - easy dependecy management for nix projects*, https://github.com/nmattia/niv, accessed: 2021-09-24.

60. *Pinned and reproducible environemt for a computational molecular modelling project*, `https://github.com/markuskowa/NixOS-QChem/tree/master/examples`, accessed: 2021-09-21.

61. T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, et al., in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, edited by F. Loizides and B. Schmidt (IOS Press, 2016), pp. 87 – 90.