Computer Physics Communications 184 (2013) 1129–1149

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

ESPResSo++: A modern multiscale simulation package for soft matter systems

Jonathan D. Halverson [a,1], Thomas Brandes [b], Olaf Lenz [a,2], Axel Arnold [b,2], Staš Bevc [c], Vitaliy Starchenko [a,3], Kurt Kremer [a], Torsten Stuehn [a,*], Dirk Reith [b,*,4]

[a] Max Planck Institute for Polymer Research, Ackermannweg 10, 55128 Mainz, Germany

[b] Fraunhofer Institute for Algorithms and Scientific Computing, Schloss Birlinghoven, 53754 Sankt Augustin, Germany

[c] National Institute of Chemistry, Hajdrihova 19, 1000 Ljubljana, Slovenia

ARTICLE INFO

Article history:
Received 14 August 2012
Received in revised form
29 November 2012
Accepted 5 December 2012
Available online 19 December 2012

Keywords:
Molecular dynamics
Monte Carlo
Simulation
Soft matter
Adaptive resolution schemes
Extensible

ABSTRACT

The redesigned Extensible Simulation Package for Research on Soft matter systems (ESPResSo++) is a free, open-source, parallelized, object-oriented simulation package designed to perform many-particle simulations, principally molecular dynamics and Monte Carlo, of condensed soft matter systems. In addition to the standard simulation methods found in well-established packages, ESPResSo++ provides the ability to perform Adaptive Resolution Scheme (AdResS) simulations which are multiscale simulations of molecular systems where the level of resolution of each molecule can change on-the-fly. With the main design objective being extensibility, the software features a highly modular C++ kernel that is coupled to a Python user interface. This makes it easy to add new algorithms, setup a simulation, perform online analysis, use complex workflows and steer a simulation. The extreme flexibility of the software allows for the study of a wide range of systems. The modular structure enables scientists to use ESPResSo++ as a research platform for their own methodological developments, which at the same time allows the software to grow and acquire the most modern methods. ESPResSo++ is targeted for a broad range of architectures and is licensed under the GNU General Public License.

1. Introduction

Molecular simulations play a central role in many areas of soft matter research. Typical examples of soft matter include polymer systems (melts, solutions, brushes, copolymers, blends ...), liquid crystals, colloids, charged fluids (from ionic liquids to polyelectrolytes), biomolecules (proteins, lipids, membranes, DNA/chromatin ...) and molecular functional systems (organic electronics, molecular sensors ...) or organic/inorganic hybrid materials like mother of pearl [1–6]. For these systems even very advanced analytic theory can only deal with idealized, rather simple conceptual questions, while experiments deal with a wide variety of synthetic and biological soft matter and ever more complex structures. This leaves a huge, growing gap, where computer simulations can contribute significantly to our understanding and link experimental findings to basic theoretical concepts. Furthermore, in materials science it is important to properly include the implications of local changes in chemical structure on macroscopic material properties. The specific properties of soft matter, namely its "softness", compared to typical low-molecular-weight materials, like metals or minerals, originates from a rather delicate interplay of energetic and entropic contributions to the free energy and a characteristic energy scale of k_BT, the thermal energy. Consequently, the typical energy density for nonbonded interactions is orders of magnitude lower than that of "hard" condensed matter, allowing for large compositional and (intra-)molecular conformational fluctuations. The low energy density and corresponding small gradients, makes molecular processes rather slow, as the relevant driving forces are originating from gradients in the already low energy density.

* Corresponding authors.
E-mail addresses: stuehn@mpip-mainz.mpg.de (T. Stuehn), dirk.reith@scai.fraunhofer.de (D. Reith).
1 Present address: Center for Functional Nanomaterials, Brookhaven National Laboratory, Upton, NY 11973, USA.
2 Present address: Institute for Computational Physics, University of Stuttgart, Pfaffenwaldring 27, 70569 Stuttgart, Germany.
3 On leave from: F. D. Ovcharenko Institute of Biocolloidal Chemistry, 42 Acad. Vernadskoho Blvd., 03142 Kyiv, Ukraine.
4 Tel.: +49 2241142746; fax: +49 2241141328.

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2012.12.004

All this has important implications for computer simulations of soft matter (see e.g., [7–11]). For many studies mid-size systems (up to about a million particles) for long times (up to milliseconds or even more) should be studied. In some cases, of course, much larger systems are needed. Also, there usually is no clear separation of time scales, making it necessary to employ so-called scale bridging or coarse-graining methodologies, where the local chemical structure is needed to properly parameterize a coarse-grained model. Such multiscale simulations are claiming an increasing fraction of soft matter simulations, as there is a shift from generic, scaling level studies to material-specific investigations. In some cases, however, it is difficult to define such a hierarchy of models and one needs to study all-atom regions and coarse-grained regions within one simulation. This is especially important for inhomogeneous systems, which is the focus of the majority of current experiments. For this, adaptive resolutions schemes like AdResS [12–15] are needed. A software system, which keeps these considerations in mind, must meet multiple requirements. Of course, the code should be scalable and fast so that it can compete with other codes such as LAMMPS [16] and NAMD [17]. In addition it has to be flexible and easily extendable by users in order to meet theoretical and experimental demands, e.g., by allowing changes in the molecular topology on-the-fly or adaptive resolution simulations of liquids. This combination of efficiency and flexibility to address completely new questions guided the definition of the design goals of our new software package ESPResSo++.

ESPResSo++ is a fully redesigned and enhanced version of the simulation software ESPResSo [18] that was developed at the Max Planck Institute for Polymer Research in Mainz, Germany, and has been used by research groups worldwide since 2005. At present, ESPResSo is maintained by the Institute for Computational Physics in Stuttgart, Germany. ESPResSo++ can be employed to perform complex simulations of many-particle systems as they are used in physics and chemistry research, in particular in soft matter physics and physical chemistry. While in principle ESPResSo++ is able to tackle all-atom simulations (e.g., with the OPLS force field) the package focuses on coarse-grained or mesoscopic simulations, where groups of atoms are represented by a single unit (usually referred to as 'superatoms'), in contrast to atomistic simulations, where a system is simulated on the atomic level. Most simulations in this field employ either the molecular dynamics (MD) method or a Monte Carlo (MC) method [19]. While ESPResSo++ is focused on the former, it also allows for so-called hybrid simulations, or even pure Monte Carlo simulations. Unlike many scientific simulation packages, ESPResSo++ is explicitly and primarily designed to be extensible, so that new algorithms and data structures can be easily included. The core is written in C++ and the user interface is written in Python. ESPResSo++ is extremely flexible and it can be used as a sandbox for the development of new methods and algorithms.

Apart from ESPResSo, a number of software packages in the field already exist and some have been maturing for almost 20 years. Some of the most widely used simulation software packages are atomistic MD simulation packages like NAMD [17], AMBER [20], DL_POLY [21], Simpatico [22] and CHARMM [23]. These packages primarily focus on many-particle simulations of models on the atomistic scale, where each atom is represented explicitly and interacts with other atoms via well-established force fields. Even though there is still research done on setting up standardized workflows for automated fitting [24–28] and optimizing specific force field parameters [29–32] and algorithms for electrostatic forces are still being improved [33,34], the general algorithmic developments are quite settled. This situation differs for mesoscopic simulations of soft matter, as they can employ a standard set of algorithms but they may be deeply optimized for that kind of simulation [7,13]. Additionally, coarse-grained model types are more diverse, and a scientist can choose amongst a number of applicable models, algorithms and system setups. Therefore, even though many coarse-grained simulations can be carried out using one of the atomistic packages, these packages are in general too inflexible. Existing software packages that are nowadays capable of simulating mesoscale soft matter systems comprise, besides ESPResSo, the IBIsCO [35], GROMACS [36], and LAMMPS [16] packages. The two latter may be used to conduct both all-atom and coarse-grained simulations, while IBIsCO is intended to be used in connection with YASP [37]. Other extensible C++ packages exist for Monte Carlo [38]. First efforts to standardize the Iterative Boltzmann Inversion mapping procedure [39,40] resulted in the CG-OPT package [41] which has been succeeded by the much more versatile VOTCA package [42]. With respect to the utilization of modern hardware and software developments, it should be mentioned that HOOMD-blue [43] performs MD simulations on GPGPU's and is similar to ESPResSo++ in that it has a Python interface with the core written in C++. MMTK [44] and SPaSM [45] are both written in C and feature a Python interface.

Traditionally, molecular simulations are performed with the same resolution for all molecules in the simulation cell. As mentioned before, this simple approach can lead to a large waste of CPU time. For instance, when studying protein molecules the solvent close to the protein should be modeled using the same level of detail as the protein but the solvent far away from the biomolecule could be treated with a coarse-grained representation or even implicitly. As solvent diffuses away from or close to the protein its representation could change smoothly from one to the other. Simulations that allow species to change their representation on-the-fly are referred to as Adaptive Resolution Schemes or AdResS [46–48]. Fig. 1 illustrates the AdResS method applied to water. To date this approach has been applied to simple fluids [46], liquid water [49,50], a solvated macromolecule [51] and C_{60} molecule [52], hydrogen bonding studies [53], and open systems [54,55]. The method has been extended to include a continuum region [56] as well as a region treated quantum mechanically [57–59]. While ESPResSo and GROMACS both provide the ability to perform AdResS simulations [52,60], these implementations were coded around the pre-existing kernels which did not efficiently support the new and unique requirements of the method. A proper, native implementation requires a code written from scratch and this was the impetus for ESPResSo++.

In this publication, we describe in detail the features of the current first official release of ESPResSo++. In Section 2 the basic design guidelines are given. The Python user interface is described in Section 3 while the C++ kernel is presented in Section 4. Section 5 consists of examples which illustrate how to extend the code. Benchmarks are given in Section 6. Finally, important information about contributing to the development of ESPResSo++ is given in the final section.

Those wishing to get started with the software should visit the following web page: http://www.espresso-pp.de. Directions for downloading and building ESPResSo++ are given in the Download section of the website and can also be retrieved from the README file in the top-most directory of the package.

2. Basic guidelines

Experience with ESPResSo, the predecessor of ESPResSo++, has shown that a few basic guidelines for the ESPResSo++ program are of utmost importance for the acceptance and efficient usage of the code by a broader scientific community.

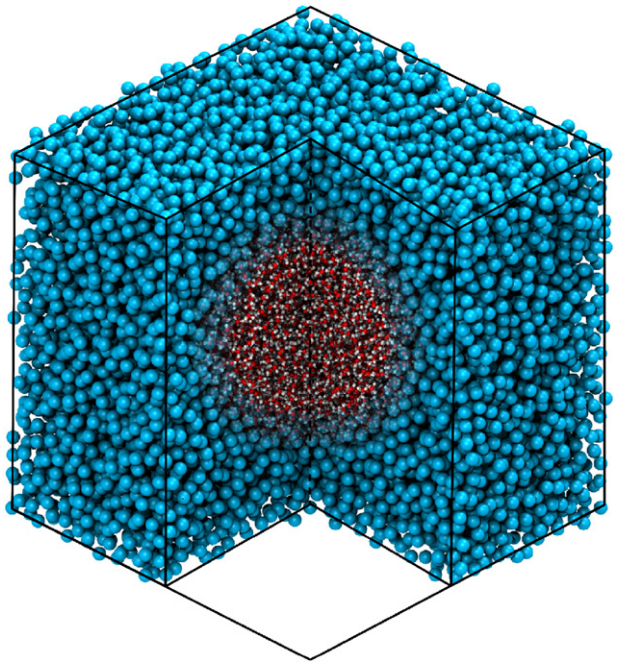

Fig. 1. A graphical illustration of AdResS for bulk liquid water. The center of the simulation box is simulated on the atomistic level while molecules away from the center are coarse-grained. A transition region is used to smoothly interpolate between the two resolutions. The molecules in the front corner region of the box are not shown to afford visualization of the interior.

Our foremost guideline is *extensibility* or the ability to easily extend both the C++ simulation engine and the Python user interface. In addition to production runs, ESPResSo++ is intended to be a sandbox for new algorithms and methods. This is supported by the program structure which was developed with not only existing algorithms in mind but also those that have yet to be conceived. In software engineering, it is commonly accepted that object-oriented modeling and programming is particularly suitable for this purpose. Therefore, ESPResSo++ is developed in the C++ programming language and employs object-oriented techniques for its design. The choice of the programming language is also reflected in the name of the package.

Performance is our second priority. This means that when conflicts between performance and extensibility arise, we choose in favor of the latter. Nevertheless, applications of ESPResSo++ are expected to run for hundreds of hours on cost-intensive high-performance computers. Therefore, it is crucial that the software be well optimized and well parallelized. ESPResSo++ is optimized towards efficient memory access and fast algorithms, and it is developed to scale well when used with a large number of processors. Even though current interpreted or byte-compiled programming languages like Java are computationally efficient in many areas of computing, for parallel high-performance numerical applications like ESPResSo++, a compiled programming language like C++ still yields a significant performance gain.

Because of the wide range of problems that will be addressed by the software and the very different demands on computing power of these problems, it is important that the software can be used on a broad range of computing platforms from standard desktop workstations to high performance supercomputers. Consequently, with *portability* in mind, ESPResSo++ does not use libraries that are not available or hard to come by on the targeted platforms. For example, the widely available message passing interface (MPI-1/2) was chosen as the library to carry out the parallelization. Furthermore, ESPResSo++ employs the C++ programming language as defined by the ANSI/ISO standard ISO/IEC 14882:2003 [61], which is portable between all major platforms. The software is designed for operating systems that comply with the POSIX standard.

The scientific problems that are handled by the software are diverse, as are the possible solutions. To allow for various strategies to handle a problem, the software is very *flexible* as to how it can be used. ESPResSo, the predecessor of ESPResSo++, uses the scripting language Tcl/Tk as a main user interface. This approach has proved to be successful and it has been decided to continue with this general practice but with a different language. ESPResSo++ employs the scripting language Python (http://www.python.org), which is amongst the most popular general-purpose multi-paradigmatic scripting languages. Python is used in many large software projects, provides numerous scientific modules and packages, is portable, has a broad user community and can be easily coupled to C++. For the coupling of C++ and Python, we rely on the widely available Boost.Python library (http://www.boost.org).

An important prerequisite for a scientist to be able to work efficiently on new applications and new algorithms with ESPResSo++ is that the source code of the program be structured clearly, that it can be understood easily, and that it is well documented. This also allows for the scientist to verify the correctness of the program. *Readability* supports maintainability, expandability and verifiability of the program.

ESPResSo++ should be as *robust* as possible. By robust we mean that the program should not become unstable if the input parameters are slightly out of the physically-reasonable bounds or the environment changes. Also, the program should be robust against changes of parameters during the run, so the physics can be altered as desired. If certain parameter combinations are formally allowed by the program, that is, they are not excluded by the documentation, then the program should support them and must not end in an uncontrolled crash. For (technically) problematic settings, the user should be warned, and in the event of a fatal error, a meaningful error message should be printed out. Note that the guideline of robustness implies that detailed and comprehensive tests for all modules of the software must be

provided and carried out. A large number of unit tests on the C++ and Python levels were used for this purpose during the development phase. Unit tests are essential for the development of a large software package to ensure not only robustness but also code correctness.

ESPResSo++ is intended to be used for scientific research as well as for industrial applications. The software is licensed under the GNU General Public License (GPL v2.0). The decision to make ESPResSo++ *freely available* should ensure that the software will always be available, and it should encourage the user community to extend ESPResSo++ and to accelerate new scientific and algorithmic developments.

3. User interface concept

Simulation software is typically provided as either a main program that takes certain input and auxiliary files (e.g., system topology, initial coordinates, force field coefficients) and produces certain output or as a library that can be used within a self-written program. ESPResSo++ employs a third way, as it uses a scripting language, namely Python, to execute and control the simulation engine which is written in C++.

Firstly, this approach allows for the software to be used in various ways, more than a fixed input file can provide without employing a scripting language. Secondly, the user of the software can employ all the other features that the scripting language provides. In particular, it allows the user to perform many tasks that are otherwise tedious to perform in an external program, like computing observables derived from several other observables, or reading in or writing out data in various formats. Lastly, simulations that are performed via the scripting language are self-documenting in the sense that the script used to execute the simulation is an exact transcript of the steps performed to obtain the results.

The choice of Python was again driven by our goal of flexibility since Python itself is a very versatile language, and being object-oriented it perfectly matches our goal of extensibility. Moreover, Python is already a well-established scripting language in the scientific community which is reflected by the vast number of existing numerical and scientific Python packages, e.g. `NumPy` and `SciPy`. While a basic understanding of Python is needed to effectively use ESPResSo++, this can be easily accomplished because Python is easy-to-learn and many books [62] and tutorials [63] are available.

Although the Python standard library is extensive, a great deal of functionality is available from numerous additional packages written by various individuals and entire communities. These packages may be pure Python or a combination of Python and compiled code which is typically written in C, C++ or Fortran. ESPResSo++ is an example of an external Python package that is a combination of Python and C++. This means that ESPResSo++ can easily take advantage of the aforementioned Python packages leading to a tightly integrated workflow from system setup to analysis. The flexible I/O concept provided by Python makes it possible to take advantage of software which is not Python based (e.g., by using TCP/IP sockets for graphical output with VMD).

The simulation core of ESPResSo++ is implemented in C++, making it possible for users to not only extend ESPResSo++ on the scripting level, but also to extend the simulation core itself by writing new C++ classes. The integration of the C++ core and the Python user interface bears two challenges. First, the classes of C++ need to be exposed as Python classes. Second, while the simulation core is often running on hundreds of processor cores, Python, like most scripting languages, can be executed only by a single core. For integrating C++ and Python, we use the Boost.Python library, which allows C++ classes to be exposed to Python with minimal effort. Currently, ESPResSo++ uses version 1.42 of the Boost libraries. The connection of the many worker cores to the single core running the script is provided by the PMI (Parallel Method Invocation) module [64], which automatically generates a serial interface for parallel method invocations following the fork-join paradigm. PMI is described in Section 4.3.

3.1. Python user interface

Typically a Python script for a simulation project has four parts: (1) import of all relevant Python modules and packages, (2) setup of system and interactions, (3) integration of the system and (4) analysis. In many cases it is useful to perform the analysis and the system integration together, or to change the setup of the system or interactions during the integration. For example, one might want to stop the integration if an observable has reached a certain value or one could switch on interactions gradually during the warm-up phase, which can often help to establish a suitable starting configuration.

In the following we give an overview of a Python script which performs a simple polymer simulation [65] in ESPResSo++. The complete script and other examples are available in the `/examples` directory. A detailed explanation of the underlying components of ESPResSo++ is given in Section 4. Comment lines begin with the # character. The first section of the script below loads various Python modules from ESPResSo++, the Python standard library and external packages:

```
import espresso
import time
import logging
import MPI
import numpy
```

In ESPResSo++ every simulation must define a system object which represents the physical system. If several simulations are to be performed simultaneously then each defines its own system object. A simple example of when this would be needed is parallel tempering [19]. A system object consists of the main components of a simulation including the simulation volume, boundary conditions, particle decomposition/communication scheme, interaction list, and the random number generator. In the code snippet below, the system object is created and its main properties are set:

```
system         = espresso.System()
system.bc      = espresso.bc.OrthorhombicBC(system.rng, box)
system.storage = espresso.storage.DomainDecomposition(system, nodeGrid, cellGrid)
system.skin    = 0.3
```

Next, a pairwise interaction, which will be computed using Verlet lists, is created. As an example we use a Lennard-Jones potential that is truncated and shifted so that it creates a purely-repulsive, continuous force. This Weeks–Chandler–Anderson (WCA) potential is often used in soft matter to represent excluded-volume interactions of hard objects such as monomer units:

```
verletWCA = espresso.VerletList(system, cutoff=rc_max)
potWCA    = espresso.interaction.LennardJones(epsilon=1, sigma=1, cutoff=2**(1.0/6), shift='auto')
interWCA  = espresso.interaction.VerletListLennardJones(verletWCA)
interWCA.setPotential(type1=0, type2=0, potential=potWCA)
```

In ESPResSo++ interactions take place between particles of specified types, which are identified by integers. In the last line of the code above, Python keywords (e.g., `type1=0`) are used to make the assignment of types clear. In this example, particles of type 0 interact according to the specified WCA interaction. Notice that units appear nowhere in the script. Because the equations of motion are solved in dimensionless form, users may enter values with whatever units they want as long as they are consistent.

A polymer composed of N monomers with a random walk structure is created by calling the `polymerRW` method of the `tools.topology` package:

```
positions, bonds = espresso.tools.topology.polymerRW(id, startpos, N, bondlength)
polymer_chain = []
for i in range(N):
  polymer_chain.append([i, positions[i], 0])
system.storage.addParticles(polymer_chain, 'id', 'pos', 'type')
system.storage.decompose()
```

The `polymerRW` method does not create the particles, but it simply returns a list of monomer positions as well as the bonds in the form of a list of tuples or ordered pairs. The positions are turned into a list of particle properties using a simple `for` loop, here fixing the identity, position and type. At this point the properties are stored as a list and only `system.storage.addParticles` actually determines how to interpret these values. A second loop could be added to create multiple chains. The `system.storage.addParticles` method adds the particles to the system and distributes the workload among the processors according to the `storage` scheme chosen above.

Next, a list of pairs is constructed to store the bonds, the bonds are added to the list, and the bonded interaction is created and added to the `system`:

```
fpl = espresso.FixedPairList(system.storage)
fpl.addBonds(bonds)
potFENE = espresso.interaction.FENE(K=30.0, r0=0.0, rMax=1.5)
interFENE = espresso.interaction.FixedPairListFENE(system, fpl, potFENE)
system.addInteraction(interFENE)
```

Lastly, the integrator is set up before the system is run 10×100 steps while computing the pressure every 100 steps:

```
integrator = espresso.integrator.VelocityVerlet(system)
...
for i in range(10):
  integrator.run(100)
  P = espresso.analysis.Pressure(system).compute()
  print P
```

The code snippet above illustrates a simple loop, although one could perform more advanced operations such as changing or adding/removing particles during the loop, modifying interactions, performing Monte Carlo moves, switching ensembles, etc. Also, it would of course be simple to terminate the loop when a certain target pressure is reached.

The visualization program VMD offers a real time connection that can be used to monitor a running simulation on-the-fly. The following code piece activates the VMD export:

```
# VMD initialization
sock = espresso.tools.vmd.connect(system)
...
# Send current positions to VMD
espresso.tools.vmd.imd_positions(system, sock)
```

This feature is useful for short production runs, classroom demonstrations, and identifying problems. In addition, the pressure and other observables could be visualized on-the-fly using `Matplotlib` [66], a powerful Python module that provides plotting functions. And with its great flexibility, `Matplotlib` is optimally suited for use with ESPResSo++.

When a simulation finishes, the code prints out the various timings and size of the neighbor list. This functionality is part of `tools.timers`. This may be used to make decisions about how to set various parameters to improve the performance of the code. For example, if a very large fraction of the time is spent in force computations or neighbor list building, then one may try varying the skin thickness.

To best way to learn ESPResSo++ is to run the example scripts which are found in `/examples`. These scripts illustrate many of the basic and advanced uses of the software. A straightforward simulation of a Lennard-Jones fluid is given in `lennard_jones.py`. For bead–spring polymer melts users should see `polymer_melt.py`. Consult `adress.py` for an AdResS simulation of tetrahedral molecules. The use of long-range forces is illustrated by `ewald.py`. The parallel tempering example `parallel_tempering.py` provides a demonstration of the use of multiple systems. Additional scripts will be added over time and users are encouraged to submit scripts which provide a clear illustration of the capabilities of ESPResSo++. While ESPResSo++ scripts tend to be longer than the input files used in other packages, the scripts make it clear how the various core components are used to carry out the simulation.

3.2. C++ developer interface

Previous experience has shown that the scripting interface alone is insufficient to provide maximal freedom. In particular, the performance of intensive numerical code pieces in the scripting language is far below the performance of these code pieces written in C++. Furthermore, it is not possible to hook functions of the scripting language into an arbitrary location in the program flow. The scripting interface has therefore been supplemented with a developers interface that enables contributors of ESPResSo++ to easily implement additional functionality directly in C++ using Boost.MPI for parallel code sections.

While the integration of C++ and Python is greatly simplified by Boost.Python, most C++ classes have explicit Python wrapper classes in ESPResSo++, which also need to be written by developers. These serve to both enhance the interface, e.g. defining properties and default values for optional arguments, and to describe the parallel interface of the classes, that is, how PMI has to invoke the methods.

The developers interface is described in detail throughout Section 4.

4. System design

C++ has emerged as the object-oriented language of choice for applications where fast performance and low-level control of memory are of great importance. It is far more difficult to gain a basic understanding of C++ in comparison to Python which a new user can become comfortable with in only a few days. Because of this ESPResSo++ was designed so that most users will never have to concern themselves with the C++ kernel. They will simply modify an existing Python example script to begin their own simulations.

Here we provide an overview of the design of the ESPResSo++ kernel and show how the various components of the software package interact with each other. This is useful for those who will need to add or modify code on the C++ level. Before beginning a major undertaking involving the ESPResSo++ kernel it is recommended that users/developers subscribe to the mailing list. Parts of this section assume a basic understanding of C++ programming. Users lacking this background are directed to a book by the original author of C++ [67] and a popular online tutorial [68].

4.1. Object-oriented design

Object-oriented development of software offers many advantages when compared with imperative programming styles. It certainly meets our major design goal of *extensibility* the best. Encapsulation, inheritance and the support for a modular architecture are the most important benefits that were used in the design of the ESPResSo++ software. This section explains in more detail the design concepts that were utilized and why these concepts were chosen.

4.1.1. Inheritance

In object-oriented programming, inheritance means the reuse of code by defining common attributes and methods within a base class that can also be used by derived classes (also called subclasses). For example, the abstract class `Storage` offers routines to send particles to and receive particles from other processors, as well as basic functionality to store the particles. These methods can be used by derived classes that determine how the particles are distributed among the available processors, for example using a spatial domain decomposition or atomic decomposition [16]. Another example is the abstract base class for interactions between particles that offers routines to compute the energy, virial, and forces. For the different potentials (e.g., Lennard-Jones, Morse) the corresponding virtual functions are overridden to compute the values prescribed by the corresponding analytical expressions.

In ESPResSo++ inheritance is also very often used in the sense of a functor. That is, the base class defines some virtual functions that will be replaced in derived classes with overridden functions. This allows, for example, the `VelocityVerlet` integrator to call the `computeForce()` method of the `Interaction` class, and other methods of the base classes `BC` (boundary conditions), `Storage`, and `Particle` where the code for the integrator itself is completely independent of the interaction potentials. In addition, nothing needs to be known about how the particles are decomposed among the processors. The integrator simply uses an iterator to traverse the particles and update their positions and velocities where appropriate.

This concept fully supports the extensibility of the software. Again, in our example, adding new derived classes of `Interaction`, `Storage` or `Particle` does not imply any changes to the algorithms used for the `VelocityVerlet` integrator.

4.1.2. Templates vs. virtual functions

The use of virtual functions in critical code sections, such as the force or energy calculation, implies an efficiency problem that can be rather well explained by the interaction classes. If the routine for the force calculation between particles becomes a virtual method, the overhead of the call becomes significant. However, the overhead can be neglected for a routine that loops over all particle pairs (or more generally tuples) in the system. Since we do not want to write this loop for each potential class, a template method is used so that the compiler generates an instance of the loop for all potentials. A similar approach is taken by HOOMD-blue [43].

The use of templates and inheritance eliminates most of the code redundancy that can be found in other simulation packages like LAMMPS [16], without adding computational overhead. This increases both the maintainability and the robustness of the software. The common practice of other packages to copy a piece of code with a similar functionality and change it slightly for a new purpose is discouraged with ESPResSo++. Instead, since classes inherit most of their behavior from their parent class, user classes typically require only a small amount of code which can be easily written with only a basic understanding of object-oriented C++.

4.1.3. Iterators

As a common and recommended practice in C++ programming we heavily exploit iterators to separate algorithms from containers. This approach makes it possible to change the internal organization of a container without needing to modify the algorithms that operate on the container. When designing ESPResSo++ it became obvious that iterators were needed when implementing algorithms that loop over particles. Containers for particles can be completely different. A simple example would be a list of particle id's. A more complicated

example would be a list that stores id's based on geometrical constraints, e.g. all particles within a specified distance of particle 42, which would not be represented explicitly as a simple list but implicitly by a predicate.

The same problem occurs for loops over particle pairs where the pairs might come from Verlet lists, cell lists or a fixed list. ESPResSo++ supports all these particle pair containers and provides iterators for them. Since these iterators are at the heart of the most time-consuming operations, namely the force or energy calculation, we achieve this flexibility not by using virtual functions, but rather by using templates as described in the previous section. However, for the force computation algorithms two template parameters are needed: one for the container of particle pairs and one for the potential. Using two parameters implies that the total number of template instantiations is the product of the number of potentials and the number of containers. Since C++ has no mechanism to instantiate such quadratic schemes implicitly and because the number of potentials is much higher than the number of containers, it was decided that the potential would remain a template parameter and the inner force loops would be explicitly written for each container. This is the one place where the need for efficiency trumped our goal of extensibility. However, the number of containers and different loop types is rather small, so the impact is minimal and flexibility is fully preserved.

4.1.4. Signals

During a simulation there might be various (physical or technical) events that might trigger software modules to act in response to these events. For example, if the identity of a particle is changed using an MC move, as in the equilibration of a solid, an event is triggered to update the particle information. A technical example would be when a resorting of particles takes place, objects that keep local references to particles have to be updated as these references are no longer valid. For such events, often called signals or publishers, we use the Boost.Signals2 library which implements routines to register or connect functions that will be called when specific events occur.

The concept of signals strongly supports the extensibility of software systems like ESPResSo++. One place where signals are used is when particles move between different cells or different processors. The particle storage provides the signal `onParticlesChanged`. A typical object that would register to this signal is the `FixedParticleList` which stores the bonding information for the particles. This object will transmit the particle's bonds to another processor when it is notified via this signal by the storage.

Another example is the `integrator` which uses signals to notify thermostats/barostats about its current state. This allows the thermostats/barostats to access particle positions, velocities or forces and influence the integration scheme. A detailed discussion of the integrator signals can be found in Section 4.8. ESPResSo++ provides a simple Python interface to disconnect an object from a signal. This makes it possible to change thermostats/barostats from within the Python script.

4.2. Integration of C++ and Python

The ESPResSo++ software relies heavily on the *interoperability* between C++ and Python. Using the Boost.Python library one can export C++ classes to Python, create C++ objects in Python and call their methods from the simulation script. One issue, however, is that different objects are linked to each other, e.g. the integrator has links to all the potential objects that are used for the force calculation. As one object might be referenced by many other objects, ESPResSo++ uses shared pointers for C++ objects to guarantee that an object will exist as long as it is referenced by any other object.

Even if C++ and Python are very close in terms of functionality, the Python user interface is more user-friendly. Therefore, after exporting a C++ class to Python, the Python class is created by deriving from the exported class and improving its interface. These enhancements usually include the following: (1) Python function arguments are named and get default values, (2) Python arguments can be higher level structures like lists or dictionaries, (3) Python methods combine different methods of the C++ classes and (4) setter and getter methods of a C++ member variable are exported as a property. The wrapping code also specifies the distribution of the class instances in a parallel environment (see Section 4.3).

4.3. PMI

From the former ESPResSo project we learned that it was very important to support the scripting of the MD simulation software in such a way that the script is completely serial and that users without any knowledge of parallelism can run the software on parallel machines. Within the ESPResSo++ project, the PMI (Parallel Method Invocation) module [64] has been developed to fulfill this requirement. PMI is illustrated in Fig. 2. PMI is a pure Python module that implements a fork-join model and has the following functionality:

- Besides the controller process, PMI also starts on the available node processes that act as workers.
- The controller process executes the serial script and only involves the workers when a PMI object is created or if a method on a PMI object is called.
- For each method on a PMI object the arguments are broadcast to the workers and the method is invoked on all processors with the same arguments.
- The controller process waits until all workers have finished their call.

This approach is only efficient if the work done by one call is sufficiently large to be executed in parallel. For MD simulations this is typically the call to run the integrator where the forces are computed. This call dominates all the other calls mainly used to setup and define the simulation system.

The PMI module has been written in a very general manner and can be used for other software packages [64]. The definition of PMI objects and PMI properties can be easily integrated in the Python wrappers that are already used for the export of C++ classes.

Users with some understanding of parallelism might prefer the execution of the Python scripts in an SPMD (single program, multiple data) manner. For this style of execution, the same Python script is executed by all processors without any broadcast of commands and arguments. This not only avoids some communication overhead but it also allows for more loosely synchronous execution. Regarding input and output and using other Python classes, users must be aware that all methods will be executed by all processors, possibly involving side effects. However, the more serious problem is that deadlocks or crashes might happen if the control flow is not the same on all processors.

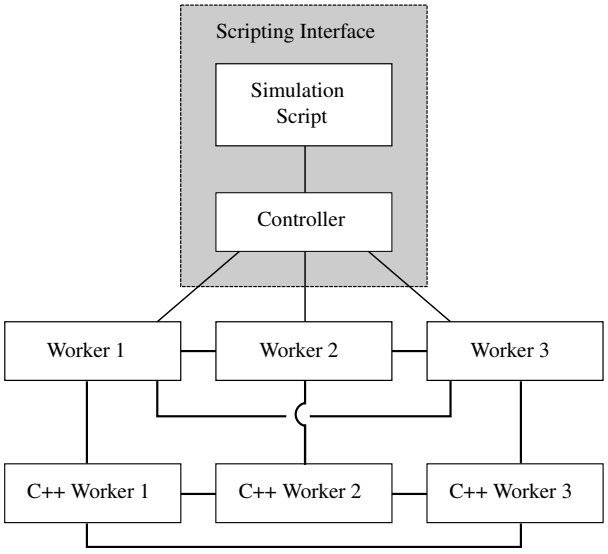

Fig. 2. Schematic illustration of PMI. The Python simulation script is executed by the controller process which divides parallel tasks among the Python workers. The C++ workers are enabled when executing critical inner loops and other time-sensitive code. The bold lines between workers indicate data communication. The Boost. MPI library is used for communication between the C++ workers while MPI for Python [69] is used for the Python workers.

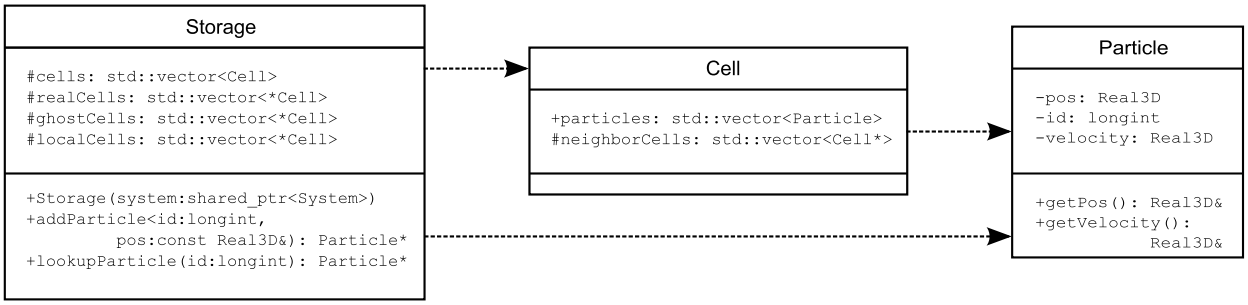

Fig. 3. Relationship between C++ classes and their principal members and methods. The Storage class contains various Cells which in turn store Particles. Arrows indicate a 'uses' relationship. The symbols indicate protected attributes (#), private attributes (−) and public methods (+).

ESPResSo++ does not explicitly support this more demanding scripting interface, but experienced users can easily make use of it by implementing PMI wrapper classes for their parallel scripts.

4.4. Error handling

Error handling is accomplished by using exceptions. The destructor concept of C++ guarantees that allocated memory will be freed for objects that are affected by the exceptions. In the case of parallel MPI applications this concept can also be utilized as long as all processors encounter the same exception. This kind of error handling can be used rather well for wrong arguments and illegal or out-of-range values. However, as soon as only a subset of processors encounter the exception, the MPI processes will no longer be synchronized leading to a deadlock. Such a deadlock can waste hours or days of valuable CPU time. For such serious problems the only solution is to abort the entire MPI application. A future version of the software will contain a more advanced error handling framework that will be based on periodically checking for exceptions and errors on all MPI processes.

4.5. Storage and domain decomposition

The storage is the container for the particles. Since the main computational effort in molecular dynamics and Monte Carlo goes into the calculation of forces or energies, the particles should be organized so as to minimize the computational effort for calculating these pair interactions, which of course depends on the nature of the interactions. For typical short-range pair interactions, such as Lennard-Jones, it is optimal to sort the particles into small spatial cells of a size just greater than the cutoff distance, so that interacting particles are only in adjacent cells [70]. Also, at processor boundaries, only a small region of these cells needs to be transferred. This organization scheme is called domain decomposition and it is described further below. Another possible scheme is atom decomposition [16], which is much more efficient if all pairs of particles interact.

The class `Storage` stores the particles in small contiguous groups called cells. Each processor is responsible for a number of these cells and the particles contained therein. In addition, a processor keeps information about particles from other nodes, which is necessary for calculating interactions. The cells and particles that belong to a processor are called 'real' while the copies of particles on other processors are called 'ghosts'. The logical structure of cells and the mapping of particles to cells is implemented in derived classes of `Storage`, with `DomainDecomposition` being an example.

Iterators for the 'real' or 'ghost' cells can be used to traverse the cells, and for a cell an iterator is available to traverse the particles of that cell (see Fig. 3). This retrieves particles in an unordered manner. In addition, there are iterators that loop over pairs of particles, which are used for example to construct the Verlet lists. The following code piece from `VerletList.cpp` illustrates this:

```
CellList cl = getSystem()->storage->getRealCells();
for (CellListAllPairsIterator it(cl); it.isValid(); ++it)
  checkPair(*it->first, *it->second);
```

The `CellListAllPairsIterator` iterates over all pairs of particles in the real cells taking into account the surrounding cells which could also be ghosts.

A particular particle can be retrieved as well, if available on the processor, by means of a global unique particle identifier. The association of the particle id to the particle data is realized by an unordered multimap provided through the Boost C++ library. This allows for fast look-ups. Particle properties are accessed by methods that deliver a reference to the corresponding particle data. This has the advantage that algorithms do not have to be changed if the layout of the particle data in memory changes in future versions.

In parallel simulations, some of the particle properties must be communicated to other processors every so many time steps. The particle properties are therefore grouped into structures according to their communication pattern. Particle communications happen in the following situations:

- If particles are exchanged between processors to optimize the organization. Such a reorganization is needed when the skin thickness is violated. If the ownership of a particle changes, all the data belonging to the particle has to be communicated to the new processor that is now responsible for the particle.
- When setting up the ghost particles the corresponding property data has to be communicated to the neighboring processors. Not all particle data is needed for the ghost particles, e.g. the velocity and force vectors are usually not communicated.
- After the particle positions are updated by the integrator, the new positions and other position-like data are communicated so that the positions of the ghost particles can be updated. In some situations (DPD simulations) the velocities are also communicated in the same way.
- After each force calculation, the forces on the ghost particles are communicated back to the processors where those particles are real.
- If a new particle is added to the system it is communicated between processors until the appropriate processor takes ownership of the particle.

Instead of transferring single particle data, the data is collected in communication buffers to allow for a single communication between a pair of processors. This avoids the rather high start-up time required by the alternative approach of sending/receiving many small messages. Communication schemes for the ghost particles can be reused for the update of ghost particles, the position update and the force communication.

The `Storage` class also provides some signals to which other routines can connect. One important signal is `onParticlesChanged` which is invoked after particles have been moved in memory or between different processors. All classes that keep pointers to particles are connected to this signal in order to update the pointers when necessary.

ESPResSo++ provides the `DomainDecomposition` storage class, which organizes the particles on a processor in cells according to a spatial domain decomposition using a regular mesh of cells, and continues this spatial decomposition also across the processor boundaries as the mapping of physical simulation space to the processors. Meloni [71] showed that the efficiency of MD simulations improves when particles close in space are stored close in memory. The `Storage` class has been written to take advantage of this fact. ESPResSo++ uses a half-shell domain decomposition scheme, which is easier to understand and modify than more advanced schemes such as the eighth-shell or midpoint method [72–75], which have a performance advantage when a large number of processors is used.

An optimization to save communication time is the introduction of a skin, similar to the well-known Verlet skin [70]. This allows particles to leave their respective cell and even processor domain for a few integration steps. Only after traveling further than a predefined skin value (or one-half this value) will the particles be redistributed among the processors. This saves expensive particle exchanges at basically no cost since the domain decomposition is typically combined with a Verlet list, which also makes use of the same skin concept. Code performance is sensitive to the skin size so the optimal value should be used. ESPResSo++ provides the `tuneSkin` method to find this value. This method may be called after the system and integrator are completely set up:

```
espresso.tools.decomp.tuneSkin(system, integrator)
```

The routine `tuneSkin` runs over different skin values, compares the execution times and determines the optimal value. This is accomplished using a Golden section [76] search.

4.6. Boundary conditions

While most simulations utilize a cubic box with periodic boundary conditions in all directions, there are many cases where other boxes are needed. For instance, in studying surface phenomena is it often better to use a box that is only periodic in two dimensions. Triclinic boxes are useful for studying crystals and for performing non-equilibrium shear simulations. And by using a truncated octahedron one can minimize the amount of solvent when studying the behavior of a hydrated biomolecule such as a protein.

In ESPResSo++ all methods and data related to the simulation box are found in the BC (boundary conditions) class. This is an abstract base class so specific box types and boundary conditions are implemented by writing a class which derives from BC. The object-oriented design of ESPResSo++ means that when a new boundary condition class is written nothing else in the code needs to be changed. How to add a new boundary condition is explained in Section 5.4.

The boundary conditions are needed when a new particle is added, the real particles are decomposed after the skin has been violated, and when the simulation box parameters are needed. The box size can be changed dynamically which is needed for constant-pressure simulations. The system object stores the boundary conditions for each simulation. Ghost particles are stored in such a way that only Euclidean distances are needed to compute pairwise nonbonded forces [77]. This avoids a relatively expensive call to the `getMinimumVector()` method of the specific boundary condition class for each pair of particles. For bonded interactions this call cannot be avoided.

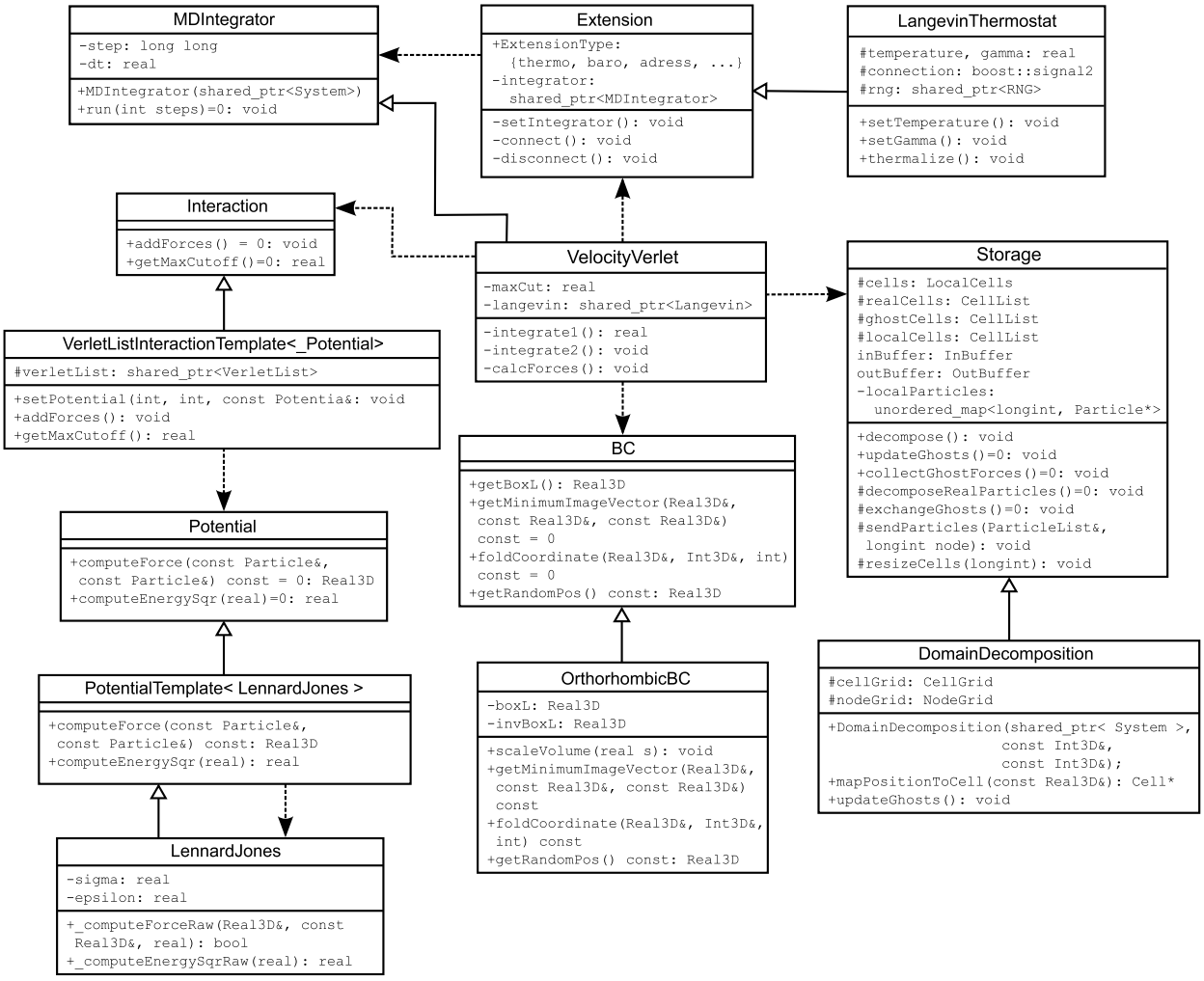

Fig. 4. Collaboration diagram of the major classes and their principal members and methods. Same symbols as Fig. 3. Solid lines and open arrows indicate the relationship between a base and derived class while the dotted lines and black arrows indicate dependencies. Lennard-Jones and the other potentials are implemented using the C++ idiom known as the curiously recurring template pattern.

4.7. Integrators

The most common numerical integration scheme in MD simulations is the velocity Verlet scheme [70]. In ESPResSo++ the `VelocityVerlet` class is derived from `MDIntegrator` which is a base class. The implementation of the `VelocityVerlet` integrator uses the classes `Interaction`, BC (boundary conditions), `Extension` and `Storage` as illustrated in Fig. 4.

The code for the integrator itself is completely independent of the interaction potentials and the decomposition of particles onto the processors. The integrator simply uses an iterator to traverse all particles that should be integrated. This example demonstrates rather well the extensibility of the software. Adding new derived classes of `Interaction`, `Storage` or `Particle` does not imply any changes to the algorithms used for the `VelocityVerlet` integrator. Additional integrators such as leap-frog, Runge–Kutta or r-RESPA may be added.

The first release of ESPResSo++ supports the constraint dynamics method SETTLE [78] which can be used to maintain rigid bonds between triangular molecules like water. Eventually the software will provide a generalized method for rigid body dynamics. The software also supports fixed particles which is useful for treating solids and for performing a primitive path analysis [79] of an entangled polymer melt.

The integrator can obtain most of the required information, like the storage and the boundary conditions, from the system object. Other properties, like thermostats and barostats, are added as integrator extensions (see Section 4.8). The integrator is therefore typically invoked as follows:

```
integrator = espresso.integrator.VelocityVerlet(system)
integrator.addExtension(langevinT)
for interval in range(100):
  integrator.run(1000)
```

Note that the `integrator.run()` method can be invoked several times in order to divide the integration into intervals. Typically, one would write out the coordinates or other observables after each interval. The example above does 100 intervals with 1000 steps per interval.

4.8. Integrator extensions

To facilitate extending the integrator, we have placed signals in critical places of the velocity Verlet integrator (e.g., before/after integrating positions, before/after force calculation, etc.). This makes it possible to add new integrator features without making any changes to the existing code of the velocity Verlet integrator. One simply creates a subclass of the abstract base class `Extension` and connects its functions to the desired signals (see Section 5.3). Thermostats, barostats, constraints, a thermodynamic force and parts of AdResS are all examples of integrator extensions.

4.9. Thermostats and barostats

Solving Newton's equations of motion for a system of particles leads to the microcanonical ensemble where the total energy and linear/angular momentum are constants. Inasmuch as experiments are typically conducted at constant temperature and/or pressure it is important to be able to simulate different ensembles as well (e.g., NVT, NPT, μVT).

In ESPResSo++ thermostats and barostats are extensions of the integrator and they are subclasses of `Extension`. This means when a new thermostat is implemented, for example, the integrator itself does not have to be modified. Users simply have to write the new code and connect to the appropriate signal (see Section 4.1.4). Currently there are several thermo/barostats implemented: `BerendsenThermostat` and `BerendsenBarostat` [80], `LangevinThermostat` and `LangevinBarostat` [81,82], Gaussian isokinetic thermostat `Isokinetic` [83], and `StochasticVelocityRescaling` [84]. When the Langevin thermostat and barostat are used together they perform Langevin dynamics in a Hoover-style extended system [82]. Additional thermostats and barostats like Andersen [85], Parrinello–Rahman [86] or a DPD thermostat [87] will be added in the next release of the software.

In order to run a simulation with a thermostat and/or barostat the user must create the corresponding thermo/barostat, set its properties and then assign it to the integrator as an extension. For the Langevin thermostat, for example, one would write:

```
langevinT = espresso.integrator.LangevinThermostat(system)
langevinT.gamma = 1.0
langevinT.temperature = 1.0
integrator.addExtension(langevinT)
```

4.10. Particle tuples

In Python a tuple is an immutable data structure used to bundle data together. It can be thought of as a list which cannot be changed after being created. In ESPResSo++, Verlet lists store a large number of tuples where each tuple contains a pair of particle id's. Lists of tuples are also employed for exclusions, i.e. lists of particle pairs that should be ignored in computing interactions. Topological information like bonds and angles are stored in a fixed list of tuples.

A fixed list of particle tuples can be created at the script level by using the standard Python syntax. ESPResSo++ provides the `FixedPairList`, `FixedTripleList`, and `FixedQuadrupleList` which store the tuples specifying the particle id's for bonds, angles and dihedrals, respectively. Tuples are added to this list through the Python interface. For instance, for adding the bonds and angle for a water molecule with the oxygen atom as particle 1:

```
bonds = espresso.FixedPairList(system.storage)
bonds.addBonds([(1, 2), (1, 3)])
angles = espresso.FixedTripleList(system.storage)
angles.addTriples([(2, 1, 3))])
```

In the next section it is shown how the bonds and angles are used to setup an interaction.

Unlike other software, ESPResSo++ allows for particles to participate in an unlimited number of bonds and angles. This is accomplished by introducing separate lists to store the bonds and angles instead of storing this information with the particles themselves.

When fixed tuple lists are distributed among the available processors, ownership of each tuple is defined by the ownership of the first particle (or second in the case of an angle). If necessary, ghost particles are used for the other particles of the tuple when computing interactions. This ensures that the necessary particle data is available for the calculation of the corresponding bonded potential.

The penalty of storing the topological information in lists as opposed to with the particles is that each communication of particles across the CPU boundaries requires the corresponding communication of tuples in all fixed lists used for the simulation. Because the storage does not track the number of fixed lists in use, to ensure that the system is up-to-date, signals are used. In order to make the correct data transfers of tuples, each list automatically connects to the following three signals of `Storage`:

```
void beforeSendParticles(ParticleList& pl, class OutBuffer& buf);
void afterRecvParticles(ParticleList& pl, class InBuffer& buf);
void onParticlesChanged();
```

When the `beforeSendParticles` signal is received the local particle list is traversed and particles that have moved to another processor have their tuples added to a buffer. When the `afterRecvParticles` signal is received, the buffer is unpacked and the new tuples are added to the local map of the processor that is responsible for a given particle. The `onParticlesChanged` signal causes the local particle list to be rebuilt from the global tuples. This is necessary when a bond or angle forms between particles which are moving to different processors. Performance tests of our standard benchmark systems have shown that this approach is as efficient as the traditional scheme where topological information is stored with the particles.

4.11. Interactions

On the Python level the user creates a set of interactions which are transferred to the C++ level. Each interaction consists of a list type (e.g., Verlet lists, cell lists, all pairs lists, fixed pairs, fixed triples, etc.) and a potential that is applied to the particle pairs in the list. The

example below illustrates how to construct a standard short-range interaction using Verlet lists and the Lennard-Jones potential:

```
potLJ   = espresso.interaction.LennardJones(epsilon=1.0, sigma=1.0, cutoff=rc, shift=False)
vl      = espresso.VerletList(system, cutoff=rc)
interLJ = espresso.interaction.VerletListLennardJones(vl)
interLJ.setPotential(type1=0, type2=0, potential=potLJ)
system.addInteraction(interLJ)
```

While Verlet lists often lead to the best performance for intermolecular interactions, occasionally cell lists are preferred. A second example shows the usage of cell lists and the Morse potential:

```
potMorse   = espresso.interaction.Morse(epsilon=10.0, alpha=2.0, rMin=0.0, cutoff=rc, shift=False)
interMorse = espresso.interaction.CellListMorse(system.storage)
interMorse.setPotential(type1=0, type2=0, potential=potMorse)
system.addInteraction(interMorse)
```

Interactions using fixed tuple lists are constructed and added to the system in an analogous manner:

```
bonds = [(1, 2), (2, 3), (3, 4)]
potFENE   = espresso.interaction.FENE(K=30.0, r0=0.0, rMax=1.5)
interFENE = espresso.interaction.FixedPairListFENE(system, bonds, potFENE)
system.addInteraction(interFENE)

angles = [(1, 2, 3), (2, 3, 4)]
potCosine   = espresso.interaction.Cosine(K=1.5, theta0=3.1415)
interCosine = espresso.interaction.FixedPairListCosine(system, angles, potCosine)
system.addInteraction(interCosine)
```

At each time step these interactions are looped over and the potential is applied to the corresponding particles to compute forces, energies and/or the contribution to the virial. The potential itself is strictly independent of the list construct. ESPResSo++ also provides the ability to exclude certain interactions as well as the ability to remove entire interactions.

The above description applies to short-range interactions. For the calculation of long-range electrostatic interactions several well-established methods exist. For periodic systems one may use Ewald summation [88], particle–particle particle-mesh (P^3M) [89], modified P^3M [33], and particle-mesh Ewald (PME) [90]. The first release of ESPResSo++ provides the Ewald summation method and in the near future P^3M will be implemented.

The Ewald summation method splits the calculation into a real and reciprocal space part [70,19]. The real space part can be calculated as a standard short-range nonbonded interaction. The reciprocal space part needs the coordinates of every particle in the system so it was implemented as a special interaction. It is worth noting that for the P^3M method the real space calculation is exactly the same, one only has to create a new potential for the reciprocal space part.

The following snippet illustrates how to add the real space interaction when the Ewald method is used:

```
vl = espresso.VerletList(system, rspacecutoff+skin)
coulombR_pot = espresso.interaction.CoulombRSpace(prefactor=1.0, alpha=1.2, cutoff=2.5)
coulombR_int = espresso.interaction.VerletListCoulombRSpace(vl)
coulombR_pot = coulombR_int.setPotential(type1=0, type2=0, potential = coulombR_pot)
system.addInteraction(coulombR_int)
```

In creating the real space potential, `prefactor` is the Coulomb prefactor, `alpha` is the Ewald parameter and `cutoff` is the real space cutoff. The second step is to create the reciprocal space interaction using cell lists:

```
ewaldK_pot = espresso.interaction.CoulombKSpaceEwald(system, coulomb_prefactor=1.0, alpha=1.2,
             kmax=5)
ewaldK_int = espresso.interaction.CellListCoulombKSpaceEwald(system.storage, ewaldK_pot)
system.addInteraction(ewaldK_int)
```

The reciprocal space cutoff is set by `kmax`. Note that the Coulomb prefactor and the Ewald parameter should have the same values for the real and reciprocal space interactions. In computing the energy a third term which is independent of particle coordinates is automatically included.

As CPU/GPGPU clusters become more available, it makes sense to have certain algorithms run on the GPU. For instance, the reciprocal space calculation in P^3M is ideally suited for this. While the software will never run solely on GPU's, over time we will enable certain parts of the code to take advantage of GPU's and possibly other accelerators.

4.12. Adaptive resolution scheme

AdResS makes it possible to have multiple regions of high and low chemical detail in one MD simulation [14,15]. For the region of interest one uses the high atomistic resolution, and for the rest of the system a lower level of detail is used. The two regions can freely interchange particles in a transition region where species change their number of degrees of freedom on-the-fly. This is illustrated in Fig. 1. Forces between two molecules are obtained with a force interpolation scheme:

$$\boldsymbol{F}_{\alpha\beta} = w(R_\alpha)w(R_\beta)\boldsymbol{F}^{atom}_{\alpha\beta} + \left[1 - w(R_\alpha)w(R_\beta)\right]\boldsymbol{F}^{cm}_{\alpha\beta},$$

Table 1
Mapping of standard classes to AdResS classes. AdResS simulations must use the AdResS classes.

Standard classes		AdResS classes
`DomainDecomposition`	→	`DomainDecompositionAdress`
`VerletList`	→	`VerletListAdress`
`FixedPairList`	→	`FixedPairListAdress`
`FixedTripleList`	→	`FixedTripleListAdress`
`FixedQuadrupleList`	→	`FixedQuadrupleListAdress`

where R_α and R_β are center-of-mass positions of molecules α and β respectively, and $w(r)$ is a weighting function that ensures a smooth transition between the two regimes.

In ESPResSo++, AdResS is implemented, like everything else, using object-oriented programming techniques. In order to run an AdResS simulation, one has to use the corresponding AdResS classes instead of the standard classes. Table 1 shows which classes should be used for AdResS. An overview of setting up and running an AdResS simulation is given below.

4.12.1. Storing atomistic particles

The atomistic particles in AdResS simulations act like an additional property of the coarse-grained particles. It is implemented in the opposite way to ESPResSo [60], where one can think of the virtual site as being an additional property of a group of atoms.

When a coarse-grained particle moves from one node to another or when its position is folded from one side of the simulation box to the other, all their atomistic particles move with it. Also, when ghost particles are created from coarse-grained particles, all its atomistic particles become ghosts too. As a consequence, atomistic particles belonging to a coarse-grained particle are always either all real or all ghost particles. This ensures that any coarse-grained particle and its corresponding atomistic particles are stored on the same node. Internally, each node stores atomistic particles in a single vector, in contrast to coarse-grained particles which are stored in multiple vectors representing cells.

Since the decomposition of particles is different in AdResS simulations, one uses the `DomainDecompositionAdress` class in the Python script:

```
system.storage = espresso.storage.DomainDecompositionAdress(system, nodeGrid, cellGrid).
```

Although AdResS provides for allowing for the creation and deletion of atomistic particles on-the-fly, in the current ESPResSo++ implementation, the number of atomistic particles is constant. In a future implementation we will evaluate possible performance gains when having a varying number of particles.

4.12.2. Mapping atomistic particles to coarse-grained particles

In order for the molecules to change their degrees of freedom when moving to a region of different resolution, the two molecule representations have to be mapped. This is done by connecting a coarse-grained particle id with a list of atomistic particle id's. Internally the mapping is handled by the `FixedTupleList` class, which uses the C++ map data structure. In the Python script one creates lists of particle id's, where the first id belongs to the coarse-grained particle and the remaining id's belong to atomistic particles. For example, the following code snippet maps the coarse-grained particles with id's 1 and 5 to atomistic particles with id's 2, 3, 4 and 6, 7, 8, respectively:

```
mapping = [[1, 2, 3, 4], [5, 6, 7, 8]]
ftpl = espresso.FixedTupleList(system.storage)
ftpl.addTuples(mapping)
system.storage.setFixedTuples(ftpl)
```

4.12.3. AdResS verlet list and force calculation

The force interpolation scheme could in principle be used across the whole simulation box, but for performance reasons it is not used in the coarse-grained region. This is accomplished by building two Verlet lists, one for the coarse-grained region and one for the atomistic–transition region. In the coarse-grained region the forces are calculated as they are in a non-AdResS simulation. Forces between molecules in the atomistic or transition region are calculated using the force interpolation scheme:

- first the positions and velocities of coarse-grained particles are overwritten by computing the center-of-mass of their atomistic particles,
- then the weights $w(r)$ are calculated using these new positions,
- the forces between two coarse-grained particles and between their atomistic particles are calculated using the weight w,
- and lastly, the forces of the coarse-grained particles are distributed to the atomistic particles.

In parallel simulations each node computes the forces for its corresponding Verlet lists. In a future ESPResSo++ implementation, spatial load-balancing of the domain will be considered, such that more nodes are available for the computationally more demanding regions.

The Verlet list must be created using the AdResS version of the class. To have the atomistic (explicit) region fixed at coordinates `x`, `y`, `z`, with size `ex` and to have the transition (hybrid) region with size `hy`, the Python code for the Verlet list would be:

```
vl = espresso.VerletListAdress(system, cutoff=rc+skin, dEx=ex, dHy=hy, adrCenter=[x, y, z])
```

The variable `vl` is a `VerletListAdress` object which is then passed as an argument to a nonbonded interaction.

Besides having the atomistic region fixed to certain coordinates in the simulation box, ESPResSo++ allows selecting an arbitrary number of particles that define the centers of *independent* atomistic regions. As these particles move around, the centers of the atomistic

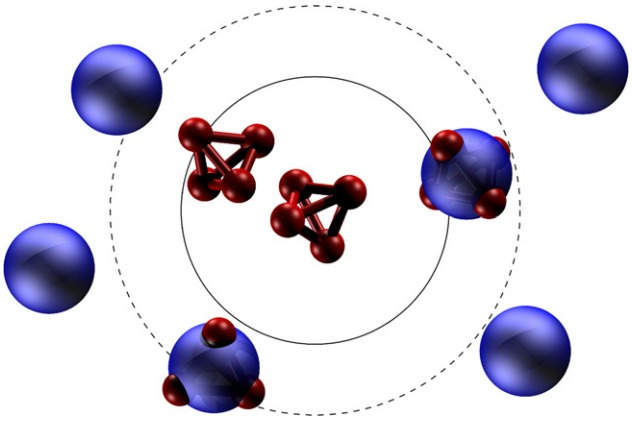

Fig. 5. A graphical illustration of AdResS where the center of the atomistic region is fixed to a particular molecule. As this molecule approaches another, the level of detail of both molecules smoothly increases. The inner circle represents the high resolution region, the area between the two circles is the transition region, and beyond the outer circle is the low resolution region.

regions move with them. This is illustrated in Fig. 5. To have the centers of atomistic regions following selected particles, one would write:

```
vl = espresso.VerletListAdress(system, cutoff=rc+skin, dEx=ex, dHy=hy, pids=[id1, id2, ...]),
```

where `[id1, id2, . . . ]` is a list of coarse-grained particle id's.

The following Python code snippet shows the setup of the interactions where a repulsive Weeks–Chandler–Andersen capped potential is used for the atomistic particle interactions and a tabulated Morse potential is used for the coarse-grained particles:

```
potWCA = espresso.interaction.LennardJonesCapped(epsilon=1.0, sigma=1.0, shift=True,
         caprad=0.27, cutoff=rca)
potMorse = espresso.interaction.Tabulated(itype=2, filename=tabMorse, cutoff=rc)
interNB = espresso.interaction.VerletListAdressLennardJonesCapped(vl, ftpl)
interNB.setPotentialAT(type1=1, type2=1, potential=potWCA)
interNB.setPotentialCG(type1=0, type2=0, potential=potMorse)
system.addInteraction(interNB)
```

Any potential class may be used for the atomistic interactions, however the coarse-grained interactions always use the tabulated class. If one wishes to use a different coarse-grained potential, a change of just one line in the source code is needed. In most cases this should not be a problem, since usually tabulated potentials are used for coarse-grained simulations. For the bonded interactions the forces are calculated across the whole simulation box, regardless of the positions of the molecules.

4.12.4. Velocity Verlet integrator and example script

As already mentioned in Section 4.8, the velocity Verlet integrator has to be extended with the AdResS extension:

```
integrator = espresso.integrator.VelocityVerlet(system)
integrator.dt = timestep
adress = espresso.integrator.Adress(system)
integrator.addExtension(adress)
```

Coarse-grained particles are propagated by the velocity Verlet integrator while the atomistic particles are propagated by the integrator extension. The atomistic time step is used across the whole simulation box.

The ESPResSo++ package contains an example Python script to run an AdResS simulation of a liquid composed of 5000 tetrahedral molecules. The script reproduces the results presented in the original work of Praprotnik et al. [47].

4.13. Analysis

ESPResSo++ is well suited for online analysis. This is due to the Python interface which allows users to not only monitor fundamental quantities like temperature and pressure but also to steer simulations based on events or measured values. An example of steering would be a script that stops a simulation when a radial distribution function has converged or one that changes force field coefficients when the distance between two particles falls below a given value. There are numerous other possibilities, many of which have been realized by users of ESPResSo [18]. This capability makes advanced workflows possible and can lead to significant savings in CPU time.

For MD simulations, an observable is a measurable quantity that is typically computed every so many time steps. In ESPResSo++ all analysis classes on the C++ level derive from the base class `Observable`. The currently available analysis classes include temperature, pressure, pressure tensor, mean-square displacement, correlation functions, center-of-mass of the system and the ability to output configurations. Particle coordinates and other properties are available to the Python script as well. The energies may be obtained directly by calling the `computeEnergy()` method of a specific interaction. Analysis involving calculations that are not time-sensitive may be done on the Python level. In Section 5.6 we show how to add a new analysis method on the C++ level.

4.14. Multiple systems

For some kind of MD applications it is useful to run multiple simulation systems in parallel and to let the systems interact with each other. One example is parallel tempering [19], also known as replica exchange, where multiple copies of the system, randomly initialized, run at different temperatures and periodically exchange information about their configurations. ESPResSo++ supports multiple interacting simulation systems running in parallel. For an example script see `/examples/parallel_tempering.py`.

Dealing with different simulation systems at the same time is done by introducing communicators and intra-communicators where the latter is used for the communication within one simulation system. Exchange of data between different simulations can be done by inter-communicators but this is currently only supported for simple cases.

Even in the case of multiple systems, ESPResSo++ offers the possibility to drive the simulation with one serial script. PMI has been extended in such a way that the processes can be grouped according to the communicators. One drawback to this approach is that the controller process executing the script must be involved with all communicators even if the controller process does not belong to the subgroup. Our current solution introduces two communicators for each subgroup, one with and one without the master process. Future work will involve implementing an alternative approach.

5. Extensibility

The object-oriented design of ESPResSo++ makes it easy to extend the code on both the C++ and Python levels. In this section we demonstrate how to implement a new interaction potential, integrator extension, boundary condition, particle property, analysis routine and file format reader/writer. Users who write code that others may benefit from are encouraged to send their contributions to the ESPResSo++ development team. If any difficulties are encountered users are encouraged to write to the mailing list which may be found through the project's website (http://www.espresso-pp.de).

5.1. Basic user concepts

Before implementing or modifying C++ code it is important that the user be familiar with a few basic concepts. Firstly, ESPResSo++ may be built in either single or double precision. This is possible because a placeholder or C++ typedef is used in the source code. Throughout the code `real` is the name of the placeholder. The compiler replaces the placeholder with either `float` or `double`, which is set in the configuration file `src/include/esconfig.hpp`. For developers extending ESPResSo++ it is important to follow this convention, otherwise changing the accuracy later will lead to errors that are very difficult to track down.

The 3-element vector is a data structure that arises frequently in particle simulation packages because it is the natural choice to store the particle coordinates, velocities, and forces. ESPResSo++ implements a specific 3-element vector class called `Real3D`. By using `Real3D` the amount of coding is reduced, mistakes become less likely, and many convenience methods are available such as the dot and cross products. The analogous structure for integers is `Int3D`.

Users should have some familiarity with PMI (see Section 4.3) which is used to implement the fork-join paradigm in the Python code. For working with the integration of C++ and Python, knowledge of the Boost.Python library is needed. When implementing nontrivial parallel code, users should be familiar with the Boost.MPI library which is used on the C++ level.

5.2. Adding a new pairwise nonbonded potential

As described in Section 4.11, the interaction potentials in ESPResSo++ are independent of the particle pair lists (i.e., Verlet lists, cell lists, fixed lists and all pairs). This means that only a few files need to be touched when adding a new pairwise nonbonded potential. On the C++ level one must write the class (header and source) and make a trivial modification to the `bindings.hpp` file, while on the Python level one must write the class and make a trivial modification to the `__init__.py` file. This amounts to creating three files and modifying only two existing files, which makes the procedure very simple and robust. While not required by the software, users are also expected to provide unit tests, which would be an additional two files (one for each on the C++ and Python levels). Documentation should also be written for new code.

The current first release of ESPResSo++ already provides a large number of potentials. When writing a new potential it is recommended that users work from an existing potential with the same number of parameters. For instance, if your new potential has two coefficients then the implementation could be based on the Lennard-Jones potential. As a specific example, let us explain how to add the Morse potential to ESPResSo++:

1. Start at the C++ level: Copy and rename `LennardJones.hpp`, `LennardJones.cpp` and `LennardJones.py` to `Morse.hpp`, `Morse.cpp` and `Morse.py`.
2. Run a search and replace within each new file to swap the names (LennardJones → Morse).
3. Rename the parameters and modify the actual expressions for the potential energy and force.
4. Finally, add the name of the potential to `src/interaction/bindings.cpp` and `__init__.py`.

New bonded and angular potentials are added in a similar manner. Users that implement a new potential are asked to send their work to the ESPResSo++ developers so that the potential can be added to the main distribution and made available to the entire community.

5.3. Adding a new integrator extension

One has to create a subclass of `Extension` to extend an integrator (e.g., velocity Verlet). The derived class should override the methods `connect()` and `disconnect()`. The `connect()` method uses Boost's bind commands to connect methods of the new class to signals located in the source file of the integrator (e.g., `VelocityVerlet.cpp`). For example, to run a method called `newMethod()` after the

force calculation, one would write:

```
_newMethod = integrator->aftCalcF.connect(boost::bind(&NewClass::newMethod, this));
```

The above would be written inside the `connect()` function to connect to the signal `aftCalcF` which is triggered after the force calculation. In the `disconnect()` method, one would call `_newMethod.disconnect()`. The easiest and recommended way to create new extensions is to take one of the existing extensions in the `integrator` directory, rename the header, source and Python files, and make the appropriate modifications.

5.4. Adding a new boundary condition

New boundary condition classes are created by deriving from the abstract base class BC as described in Section 4.6. This has already been done e.g. for orthorhombic boxes with the class `OrthorhombicBC`. Another example would be to create a new boundary condition that is nonperiodic only in the z-direction. Note that for this case the proper implementation would involve writing a C++ template class to handle the other directions as well. To do this one would copy and rename the header and source files of `OrthorhombicBC` to something like `ZnonperiodicBC`. One would then need to make the appropriate modifications to the methods `getMinimumImageVector`, `foldCoordinate` and related methods. To export the new code, the corresponding Python file would also have to be created. Trivial modifications would be needed to `__init__.py` and the C++ bindings file. This procedure may be used to add classes that support triclinic, truncated octahedron, rhombic dodecahedron, mirror, or other boundary conditions.

5.5. Adding a new particle property

The `Particle` class is found in `Particle.hpp`. All particles in ESPResSo++ have the properties of id, type, mass, charge, position, velocity and force. However, for efficiency reasons not all of these properties are up-to-date at all times. For instance, for ghost particles by default only their positions are reliable, but not their velocities. Additionally, after their forces are updated they are communicated to the processor where they are real particles. As explained in Section 4.5, it is essential to know how a new particle property needs to be communicated.

Consider adding the radius of a particle as a new property. This can be done by adding the data member `real radius` to the `ParticleProperties` structure. This will ensure that the `radius` is always up-to-date in the event that the user wants to vary it during a simulation. One must also write the trivial getter and setter methods as well as assigning a default value to `radius` in the `Particle` constructor. The communication of the property is then automatically performed without further action.

5.6. Adding a new analysis method

Here we demonstrate how to add a new analysis class which can be used from the Python interface. As described in Section 4.13 all analysis classes in ESPResSo++ derive from the base class `Observable`. `Observable` provides a virtual method called `compute()` which is implemented in the derived class. For the class responsible for computing the center-of-mass (COM) of the system, for instance, its `compute()` method must be written to carry out this calculation. In the Python script the appropriate analysis object must be created e.g., `com = analysis.CenterOfMass(system)`. Later in the script the compute method should be called e.g., `CM = com.compute()`. When this method is called, the PMI controller instructs all the workers to call their compute methods on their local data. This amounts to getting the cells containing real particles from the storage and then iterating over the particles while summing their mass and mass-weighted position vector. The reduction, i.e. the summation of the local contributions and the transmission to the script executing process is handled by PMI.

The calculation of the COM or the temperature of a system is fairly straightforward. However, computing the pressure or pressure tensor is more involved. The `Pressure` and `PressureTensor` classes illustrate how one can use the short-range interaction list, tensor class and Verlet list to compute some quantity. These two examples demonstrate many of the concepts and make it easy for a user to create their own analysis class.

5.7. Adding a new file format reader/writer

The above examples illustrate how to extend the C++ code. Most users will never need to do this but will instead work with the Python code where a fair amount of functionality is written. For instance, the code for reading and writing files of a specific format is written in Python. This was done because Python, being a scripting language, is far better suited for processing text files than C++ which is by comparison cumbersome and less compact. The code for handling file formats is located in `src/tools/convert/`. One typically needs to extract the topology, positions and velocities which can be accomplished with basic Python methods. Note that the newly written format methods can be used immediately without compiling since the Python code is dynamically interpreted.

6. Benchmarks

While the ESPResSo++ simulation package has been designed with extensibility as its top priority, we demonstrate in this section that it shows comparable performance to well-established codes on both a single processor and in parallel. The reader should be aware that benchmark results can vary widely and the numbers we report may change significantly when different compilers, machines and libraries are used.

We consider the following benchmark systems:

- Benchmark 1: Lennard-Jones fluid with $N = 32\,768$, $\rho = 0.8442/\sigma^3$, $r_c = 2.5\,\sigma$, $\Delta t = 0.005\,\tau$, and the velocity Verlet integration scheme. These choices realize the NVE ensemble. The average temperature was roughly $0.8\,\epsilon/k_B$.

- Benchmark 2: Same as Benchmark 1 except with $N = 884\,736$.
- Benchmark 3: Semiflexible bead–spring polymer melt [65] with $N = 320\,000$, $\rho = 0.85/\sigma^3$, $T = 1\,\epsilon/k_B$, $r_c = 2^{1/6}\,\sigma$, $\Delta t = 0.01\,\tau$, velocity Verlet integration scheme and a Langevin thermostat with a friction coefficient of $1\,\tau^{-1}$. Bonds were treated with the `FENE` interaction and `AngularCosine` was used for the angles. The initial configuration was an equilibrated melt of 1600 ring polymers with 200 monomers per ring [91].
- Benchmark 4: An AdResS simulation of 5000 tetrahedron molecules at $T = 1\,\epsilon/k_B$ (see Section 4.12 and Ref. [47] for complete details).

Benchmark 1 considers a simple fluid and it demonstrates the ability of the code to handle short-range interactions where each particle has several tens (≈55) of neighbors. The second benchmark considers a very large system which allows for a demonstration of the scaling performance of ESPResSo++ on hundreds to thousands of cores. Benchmark 3 is a polymer melt where each monomer has a small number of neighbors (≈6) and bonded interactions are used to maintain the topology of chains. Benchmark 4 is an AdResS simulation of tetrahedron molecules.

All codes were built in double precision at optimization level `O3` using an MPI compiler, even for the single CPU tests. The benchmarks were carried out using ESPResSo++ 1.0.0, ESPResSo 3.0.1 (with minimal activated features), and LAMMPS (lammps-20Jun11). Each simulation was run for $P \times 10^3$ steps, where P is the number of cores. This choice caused each benchmark to run for at least 1–2 min where a sufficiently large number of neighbor list updates took place. The skin was optimized for each code and for each value of P. This was done by varying the skin from 0.05 to 1.2 σ in increments of 0.05 σ and reporting the lowest time. Only the time spent integrating the equations of motion was counted i.e., time spent reading the initial coordinates and velocities was not included. No analysis was performed. Note that all three codes were found to give identical results for Benchmark 1 for the first tens of steps before round-off errors caused the trajectories to diverge.

6.1. Benchmark results

Results for Benchmarks 1 and 2 are shown in Fig. 6 for ESPResSo++, ESPResSo and LAMMPS on a standard Infiniband Linux cluster with Intel Xeon CPU X5570 2.93 GHz, 8192 kB cache, 8 cores per node, Intel C/C++ compiler, Suse Linux SLES 11 and Intel MPI 4.0.0. For both system sizes ESPResSo++ and ESPResSo show nearly the same performance while LAMMPS outperforms both packages. When 64 cores are used LAMMPS is 1.5x faster for Benchmark 1 and 1.3x faster for Benchmark 2 in comparison to ESPResSo++. The inset of Fig. 6(b) shows that the same trend holds on a high-performance Linux cluster. This performance advantage is important for long simulations but for simulations that run for only a few hours it is minor.

The polymer melt benchmarks are shown in Fig. 7. Once again LAMMPS enjoys a performance advantage but for the largest number of cores, the speed-up is minimal. ESPResSo++ consistently outperforms ESPResSo for this benchmark where particles exchange their neighbors frequently.

The AdResS benchmark results shown in Fig. 8 indicate that the code shows an approximate linear scaling up to about 50 cores or 400 particles per core. This is a demonstration of good performance since high parallel efficiencies are typically achieved with hundreds or even a few thousand particles per core.

6.2. Analysis of benchmark results

To understand the reasons for the difference in performance between ESPResSo++ and LAMMPS we measured the time that each code spends performing the most costly operations which are the force calculations, updating the Verlet list and communicating particle data. The main reasons for the difference in performance appears to be that LAMMPS builds the neighbor lists faster and it spends less time communicating particle data. The advantage in building neighbor lists arises from the fact that LAMMPS supports cell sizes smaller than the cutoff and specifically it uses cell sizes that are one-half those of ESPResSo++. This means that fewer particle pairs have their separation distance checked when building Verlet lists or when using cell lists. We identified that ESPResSo++ shows comparable performance to LAMMPS in computing pairwise forces. LAMMPS requires 32 bytes for each particle since it stores three double-precision numbers for the particle position and one 8-byte pointer to the array of particle positions. The choice to store the pointer means that the compiler cannot optimize the code as well as it can for ESPResSo++. LAMMPS also uses pointers to get the coefficients for the interaction potentials. This leads to a slight slowdown in comparison to storing the value with the interaction as is done in ESPResSo++. Future work will involve generalizing the cell scheme and improving on the communication of particle data.

Another reason for the difference in performance between the two codes is the storage layout of the particle data. While LAMMPS stores particle data for each property separately, ESPResSo++ stores all particle data of each particle contiguously in one record. The former approach has advantages when algorithms work only on a few properties (e.g., force calculation, integration), the latter layout is favorable for algorithms working on nearly all particle data (e.g., moving particles between cells). In both cases this is due to better cache usage. As most of the time-consuming algorithms in MD simulations belong to the first category, LAMMPS has the better solution. Therefore, we will consider this storage layout in our next version. Due to the object-oriented design we can do this without rewriting the algorithms that work on particle data.

7. Conclusions

With its highly extensible design, Python interface, and support of AdResS, the ESPResSo++ simulation package is a new, advanced tool which can be used to study of a wide range of soft matter systems. The modular design of its C++ kernel makes it easy to add new algorithms or modify existing ones. The Python interface of the package provides tremendous flexibility and the ability to quickly set up a simulation. The current first release of the software supports parallel molecular dynamics simulations in the NVE, NPT or NVT ensembles, multiple interacting systems, constraint dynamics, electrostatics, molecular force fields, excluded interactions, dynamic bonding and AdResS. ESPResSo++ is the first software package to provide a native implementation of AdResS, making it possible to quickly set up and run an AdResS simulation.

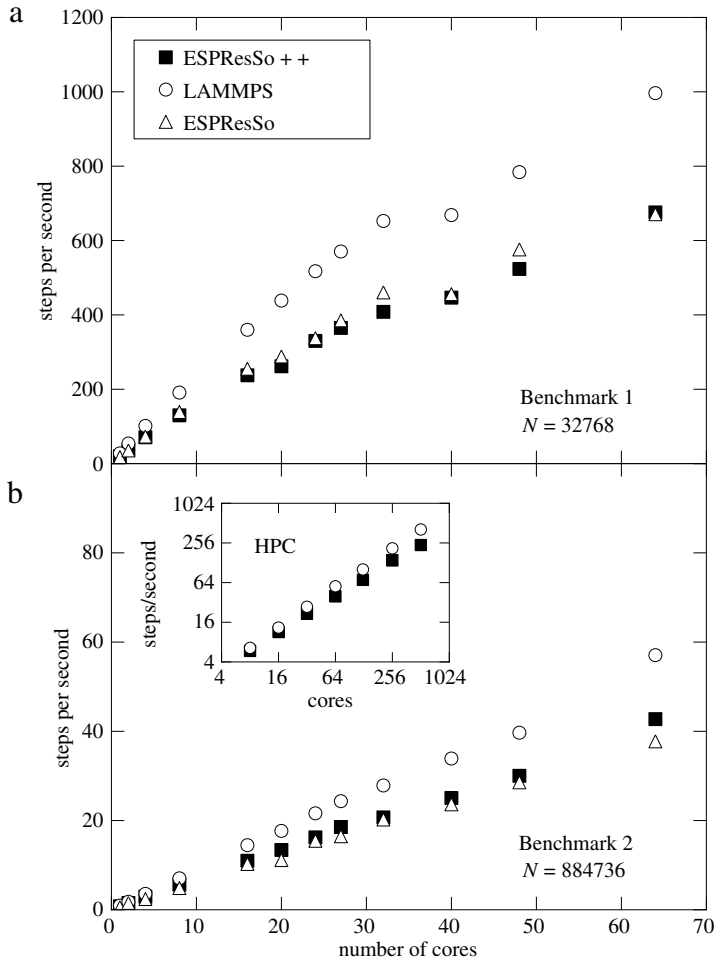

Fig. 6. Performance of ESPResSo++, LAMMPS and ESPResSo on a standard Linux cluster for a Lennard-Jones fluid with (a) $N = 32\,768$ and (b) $N = 884\,736$. The inset of (b) shows the performance of ESPResSo++ and LAMMPS for $N = 884\,736$ (Benchmark 2) on an high-performance Linux cluster (HPC) with a fast (QDR Infiniband) tree structure.

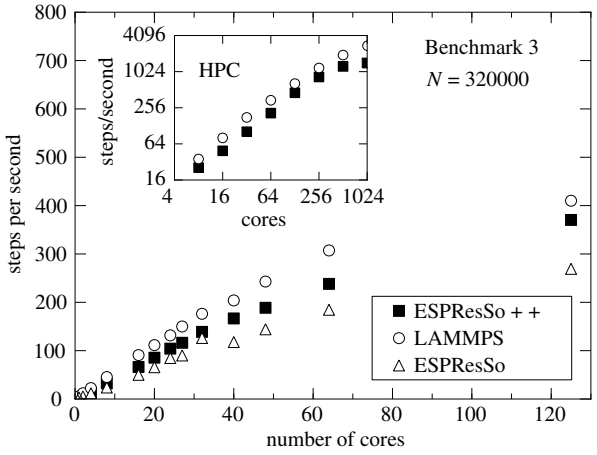

Fig. 7. Performance of ESPResSo++, LAMMPS and ESPResSo for polymer melt with $N = 320\,000$ on a standard Linux cluster. The inset shows the results for ESPResSo++ and LAMMPS on an HPC Linux cluster with a fast (QDR Infiniband) tree structure.

The greatest advantage of open-source software is that individual efforts move the entire community forward. All members of the ESPResSo++ community are encouraged to participate in the application and growth of the software. In this way we expect the software to acquire the most modern algorithms. Additionally, the scientific Python community is growing rapidly and because ESPResSo++ can take advantage of these packages, ESPResSo++ effectively grows along with it. Users who have extended the software on either the C++ or Python level are therefore strongly encouraged to contact the ESPResSo++ development team so that the new code can be added to the main distribution. To support this goal a collaborative development platform is maintained at http://www.espresso-pp.de.

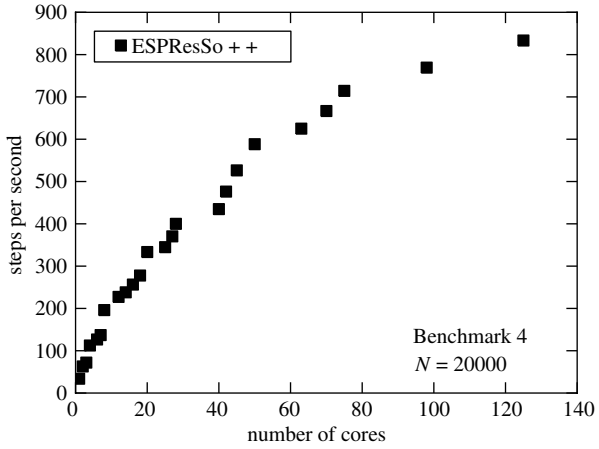

Fig. 8. Strong scaling behavior of a 20 000 particle AdResS simulation (Benchmark 4) with ESPResSo++.

Acknowledgments

This work was funded by a research grant awarded to the Fraunhofer Institute SCAI and the MPI for Polymer Research. Victor Rühle is acknowledged for development efforts involving the `ParticleGroup` class. Konstantin Koschke implemented the stochastic velocity rescaling thermostat. Christoph Junghans is responsible for the `cmake` build system. We thank Anton Schüller for participation in the early stages of development, and Luigi Delle Site, Matej Praprotnik, Christoph Junghans, Simon Poblete and Sebastian Fritisch for discussions regarding AdResS. Tristan Bereau contributed Python code during first phase of the project. This work benefited from detailed discussions about MD simulation and its implementation with Berk Hess. One of us (S.B.) acknowledges the hospitality of the Max Planck Institute for Polymer Research where AdResS was implemented. Research carried out in part at the Center for Functional Nanomaterials, Brookhaven National Laboratory, which is supported by the U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. DE-AC02-98CH10886.

References

[1] A. Dobrynin, M. Rubinstein, Theory of polyelectrolytes in solutions and at surfaces, Progress in Polymer Science 30 (11) (2005) 1049–1118.
[2] F. Bates, M. Hillmyer, T. Lodge, C. Bates, K. Delaney, G. Fredrickson, Multiblock polymers: panacea or Pandora's box? Science 336 (6080) (2012) 434–440.
[3] T. Boudou, T. Crouzier, K. Ren, G. Blin, C. Picard, Multiple functionalities of polyelectrolyte multilayer films: New biomedical applications, Advanced Materials 22 (4) (2010) 441–467.
[4] F. Hoeben, P. Jonkheijm, E. Meijer, A. Schenning, About supramolecular assemblies of π-conjugated systems, Chemical Review 105 (2005) 1491–1546.
[5] R. Kroon, M. Lenes, J.C. Hummelen, P.W.M. Blom, B. de Boer, Small bandgap polymers for organic solar cells (polymer material development in the last 5 years), Polymer Reviews 48 (3) (2008) 531–582.
[6] C.J. Brabec, M. Heeney, I. McCulloch, J. Nelson, Influence of blend microstructure on bulk heterojunction organic photovoltaic performance, Chemical Society Reviews 40 (3) (2011) 1185–1199.
[7] K. Kremer, F. Müller-Plathe, Multiscale simulation in polymer science, Molecular Simulation 28 (2002) 729–750.
[8] N. Attig, K. Binder, H. Grubmüller, K. Kremer (Eds.), Computational Soft Matter: From Synthetic Polymers to Proteins, in: NIC Lecture Notes, vol. 23, Forschungszentrum Jülich, 2004.
[9] G.A. Voth (Ed.), Coarse-Graining of Condensed Phase and Biomolecular Systems, CRC Press, Boca Raton, Florida, 2008.
[10] C. Holm, K. Kremer (Eds.), Advanced Computer Simulation Approaches for Soft Matter Science I–III, in: Series: Advances in Polymer Science, vols. 173, 185, 221, Springer, Berlin, 2005–2009.
[11] C. Peter, K. Kremer, Multiscale simulation of soft matter systems, Faraday Discussions 144 (2010) 9–24.
[12] M. Praprotnik, L.D. Site, K. Kremer, Multiscale simulation of soft matter: from scale bridging to adaptive resolution, Annual Review of Physical Chemistry 59 (1) (2008) 545–571.
[13] M. Praprotnik, C. Junghans, L. Delle Site, K. Kremer, Simulation approaches to soft matter: generic statistical properties vs. chemical details, Computer Physics Communications 179 (2008) 51–60.
[14] S. Poblete, M. Praprotnik, K. Kremer, L.D. Site, Coupling different levels of resolution in molecular simulations, Journal of Chemical Physics 132 (11) (2010) 114101.
[15] M. Praprotnik, S. Poblete, K. Kremer, Statistical physics problems in adaptive resolution computer simulations of complex fluids, Journal of Statistical Physics (2011) 946–966.
[16] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1995) 1–19.
[17] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, Scalable molecular dynamics with NAMD, Journal of Computational Chemistry 26 (2005) 1781–1802.
[18] H.J. Limbach, A. Arnold, B.A. Mann, C. Holm, ESPResSo — an extensible simulation package for research on soft matter systems, Computer Physics Communications 174 (9) (2006) 707–727.
[19] D. Frenkel, B. Smit, Understanding Molecular Simulation, second ed., Academic Press, San Diego, 2002.
[20] D.A. Case, T.A. Darden, T.E. Cheatham III, C.L. Simmerling, J. Wang, R.E. Duke, R. Luo, R.C. Walker, W. Zhang, K.M. Merz, B. Roberts, B. Wang, S. Hayik, A. Roitberg, G. Seabra, I. Kolossvai, K.F. Wong, F. Paesani, J. Vanicek, J. Liu, X. Wu, S.R. Brozell, T. Steinbrecher, H. Gohlke, Q. Cai, X. Ye, J. Wang, M.-J. Hsieh, G. Cui, D.R. Roe, D.H. Mathews, M.G. Seetin, C. Sagui, V. Babin, T. Luchko, S. Gusarov, A. Kovalenko, P.A. Kollman, AMBER 11, University of California, San Francisco, 2010.
[21] W. Smith, I.T. Todorov, A short description of DL_POLY, Molecular Simulation 32 (2006) 935–943.
[22] Simpatico — Simulation Package for Polymer and Molecular Liquids, 2011. URL http://www.cems.umn.edu/research/morse/code/simpatico/home.php.
[23] B.R. Brooks, C.L. B III, A.D. Mackerell, L. Nilsson, R.J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S.B.A. Caflisch, L. Caves, Q. Cui, A.R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kuczera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R.W. Pastor, C.B. Post, J.Z. Pu, M. Schaefer, B. Tidor, R.M. Venable, H.L. Woodcock, X. Wu, W. Yang, D.M. York, M. Karplus, CHARMM: the biomolecular simulation program, Journal of Computational Chemistry 30 (2009) 1545–1615.
[24] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, J. Myers, Examining the challenges of scientific workflows, Computer 40 (12) (2007) 24.
[25] M. Hülsmann, T. Köddermann, J. Vrabec, D. Reith, Grow: a gradient-based optimization workflow for the automated development of molecular models, Computer Physics Communications 181 (2010) 499–513.
[26] M. Hülsmann, J. Vrabec, A. Maaß, D. Reith, Assessment of numerical optimization algorithms for the development of molecular models, Computer Physics Communications 181 (2010) 887–905.

[27] B. Waldher, J. Kuta, S. Chen, N. Henson, A.E. Clark, ForceFit: a code to fit classical force fields to quantum mechanical potential energy surfaces, Journal of Computational Chemistry 31 (12) (2010) 2307–2316.
[28] D. Reith, K.N. Kirschner, A modern workflow for force-field development—bridging quantum mechanics and atomistic computational models, Computer Physics Communications 182 (2011) 2184–2191.
[29] O. Guvench, A.D. MacKerell Jr., Automated conformational energy fitting for force-field development, Journal of Molecular Modeling 14 (8) (2008) 667–679.
[30] B. Eckl, J. Vrabec, H. Hasse, Set of molecular models based on quantum mechanical ab initio calculations and thermodynamic data, Journal of Physical Chemistry B 112 (2008) 12710–12721.
[31] M. Hülsmann, T.J. Müller, T. Köddermann, D. Reith, Automated force field optimisation of small molecules using a gradient-based workflow package, Molecular Simulation 36 (2011) 1182–1196.
[32] T. Köddermann, K.N. Kirschner, J. Vrabec, M. Hülsmann, D. Reith, Liquid–liquid equilibria of dipropylene glycol dimethyl ether and water by molecular dynamics, Fluid Phase Equilibria 310 (2011) 25–31.
[33] V. Ballenegger, J.J. Cerda, O. Lenz, C. Holm, The optimal P3M algorithm for computing electrostatic energies in periodic systems, Journal of Chemical Physics 128 (3) (2008) 034109.
[34] V. Ballenegger, A. Arnold, J.J. Cerda, Simulations of non-neutral slab systems with long-range electrostatic interactions in two-dimensional periodic boundary conditions, Journal of Chemical Physics 131 (9) (2009) 094107.
[35] H. Karimi-Varzaneh, H. Qian, X. Chen, P. Carbone, F. Müller-Plathe, IBIsCO: a molecular dynamics simulation package for coarse-grained simulation, Journal of Computational Chemistry 32 (7) (2011) 1475–1487.
[36] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation, Joural of Chemical Theory Computer 4 (2008) 435–447.
[37] F. Müller-Plathe, YASP: a molecular simulation package, Computer Physics Communications 78 (1993) 77–94.
[38] M. Lund, M. Trulsson, B. Persson, Faunus: an object-oriented framework for molecular simulation, Source Code for Biology and Medicine 3 (2008) 1–8.
[39] D. Reith, M. Pütz, F. Müller-Plathe, Deriving effective mesoscale potentials from atomistic simulations, Journal of Computational Chemistry 24 (13) (2003) 1624–1636.
[40] R. Faller, D. Reith, Properties of poly(isoprene): model building in the melt and in solution, Macromolecules 36 (2003) 5406–5414.
[41] D. Reith, H. Meyer, F. Müller-Plathe, CG-OPT: a software package for automatic force field design, Computer Physics Communications 148 (3) (2002) 299–313.
[42] V. Ruehle, C. Junghans, A. Lukyanov, K. Kremer, D. Andrienko, Versatile object-oriented toolkit for coarse-graining applications, Journal of Chemical Theory Computer 5 (2009) 3211–3223.
[43] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008) 5342–5359.
[44] K. Hinsen, The molecular modeling toolkit: a new approach to molecular simulations, Journal of Computational Chemistry 21 (2000) 79–85.
[45] D.M. Beazley, P.S. Lomdahl, Building flexible large-scale scientific computing applications with scripting languages, in: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN USA, 1997.
[46] M. Praprotnik, L.D. Site, K. Kremer, Adaptive resolution molecular-dynamics simulation: changing the degrees of freedom on the fly, Journal of Chemical Physics 123 (22) (2005) 224106–224114.
[47] M. Praprotnik, L.D. Site, K. Kremer, Adaptive resolution scheme for efficient hybrid atomistic-mesoscale molecular dynamics simulations of dense liquids, Physical Review E 73 (2006).
[48] M. Praprotnik, K. Kremer, L.D. Site, Fractional dimensions of phase space variables: a tool for varying the degrees of freedom of a system in a multiscale treatment, Journal of Physics A 40 (15) (2007) F281–F288.
[49] M. Praprotnik, S. Matysiak, L.D. Site, K. Kremer, C. Clementi, Adaptive resolution simulation of liquid water, Journal of Physics: Condensed Matter 19 (29) (2007) 292201.
[50] M. Praprotnik, S. Matysiak, L.D. Site, K. Kremer, C. Clementi, Corrigendum: adaptive resolution simulation of liquid water, Journal of Physics: Condensed Matter 21 (49) (2009) 499801.
[51] M. Praprotnik, L.D. Site, K. Kremer, A macromolecule in a solvent: Adaptive resolution molecular dynamics simulation, Journal of Chemical Physics 126 (13) (2007) 134902.
[52] S. Fritsch, C. Junghans, K. Kremer, Structure formation of toluene around C60: Implementation of the adaptive resolution scheme (AdResS) into GROMACS, Journal of Chemical Theory Computer 8 (2) (2012) 398–403.
[53] B.P. Lambeth Jr., C. Junghans, K. Kremer, C. Clementi, L. Delle Site, Communication: on the locality of hydrogen bond networks at hydrophobic interfaces, Journal of Chemical Physics 133 (2010) 221101.
[54] D. Mukherji, N.F.A. van der Vegt, K. Kremer, L. Delle Site, Kirkwood-Buff analysis of liquid mixtures in an open boundary simulation, Journal of Chemical Theory Computer 8 (2) (2012) 375–379.
[55] S. Fritsch, S. Poblete, C. Junghans, G. Ciccotti, L.D. Site, K. Kremer, Adaptive resolution molecular dynamics simulation through coupling to an internal particle reservoir, Physical Review Letters 108 (2012) 170602.
[56] R. Delgado-Buscalioni, K. Kremer, M. Praprotnik, Concurrent triple-scale simulation of molecular liquids, Journal of Chemical Physics 128 (2008) 114110.
[57] A.B. Poma, L.D. Site, Classical to path-integral adaptive resolution in molecular simulation: Towards a smooth quantum-classical coupling, Physical Review Letters 104 (2010) 250201.
[58] A.B. Poma, L. Delle Site, Adaptive resolution simulation of liquid para-hydrogen: testing the robustness of the quantum-classical adaptive coupling, Physical Chemistry Chemical Physics 13 (22) (2011) 10510–10519.
[59] R. Potestio, L. Delle Site, Quantum locality and equilibrium properties in low-temperature parahydrogen: a multiscale simulation study, Journal of Chemical Physics 136 (2012) 054101.
[60] C. Junghans, S. Poblete, A reference implementation of the adaptive resolution scheme in ESPResSo, Computer Physics Communications 181 (8) (2010) 1449–1454.
[61] International Organization for Standardization, ISO/IEC 14882:2003: Programming languages — C++, International Organization for Standardization, Geneva, Switzerland, 2003.
[62] H.P. Langtangen, A Primer on Scientific Programming with Python, Springer, Berlin, 2011.
[63] Official Python Tutorial, 2012. URL http://docs.python.org/tutorial/.
[64] O. Lenz, PMI — parallel method invocation, in: G. Varoquaux, S. van der Walt, J. Millman (Eds.), Proceedings of the 8th Python in Science Conference, Pasadena, CA USA, 2009, pp. 48–50.
[65] K. Kremer, G.S. Grest, Dynamics of entangled linear polymer melts: a molecular-dynamics simulation, Journal of Chemical Physics 92 (8) (1990) 5057–5086.
[66] Matplotlib plotting library, 2012. URL http://matplotlib.sourceforge.net.
[67] B. Stroustrup, The C++ Programming Language, Addison-Wesley, Boston, 2000.
[68] C++ Tutorial, 2012. URL http://www.cplusplus.com/doc/tutorial/.
[69] MPI for Python, 2012. URL http://mpi4py.scipy.org/.
[70] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford University Press, New York, 1987.
[71] S. Meloni, M. Rosati, L. Colombo, Efficient particle labeling in atomistic simulations, Journal of Chemical Physics 126 (2007) 121102.
[72] K.J. Bowers, R.O. Dror, D.E. Shaw, Zonal methods for the parallel execution of range-limited N-body simulations, Journal of Computational Physics 221 (1) (2007) 303–329.
[73] D. Shaw, A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions, Journal of Computational Chemistry 26 (13) (2005) 1318–1328.
[74] K. Bowers, R. Dror, D. Shaw, Overview of neutral territory methods for the parallel evaluation of pairwise particle interactions, in: A. Mezzacappa (Ed.), Scientific Discovery Through Advanced Computing, SciDAC 2005, in: Journal of Physics Conference Series, vol. 16, US Dept Energy Off Sci, 2005, pp. 300–304. Conference of Scientific Discovery through Advanced Computing (SciDAC 2005), San Francisco, CA, JUN 26–30, 2005.
[75] K. Bowers, R. Dror, D. Shaw, The midpoint method for parallelization of particle simulations, Journal of Chemical Physics 124 (2006) 184109.
[76] D.E. Knuth, The Art of Computer Programming, Vol. 3, second ed., Addison-Wesley, Boston, 1997.
[77] D.C. Rapaport, Large-scale molecular dynamics simulation using vector and parallel computers, Computer Physics Reports 9 (1988) 1–53.
[78] S. Miyamoto, P.A. Kollman, SETTLE: an analytical version of the shake and rattle algorithm for rigid water models, Journal of Computational Chemistry 13 (1992) 952–962.
[79] R. Everaers, S.K. Sukumaran, G.S. Grest, C. Svaneborg, A. Sivasubramanian, K. Kremer, Rheology and microscopic topology of entangled polymeric liquids, Science 303 (2004) 823–826.
[80] H.J.C. Berendsen, J.P.M. Postma, W.F. van Gunsteren, A.D. Nola, J.R. Haak, Molecular dynamics with coupling to an external bath, Journal of Chemical Physics 81 (1984) 3684–3690.

[81] T. Schneider, E. Stoll, Molecular-dynamics study of a three-dimensional one-component model for distortive phase transitions, Physical Review B 17 (1978) 1302.
[82] D. Quigley, M.I.J. Probert, Langevin dynamics in constant pressure extended systems, Journal of Chemical Physics 120 (24) (2004) 11432–11441.
[83] D.J. Evans, S. Sarman, Equivalence of thermostatted nonlinear responses, Physical Review E 48 (1993) 65–70.
[84] G. Bussi, D. Donadio, M. Parrinello, Canonical sampling through velocity rescaling, Journal of Chemical Physics 126 (2007) 014101.
[85] H. Andersen, Molecular dynamics at constant pressure and/or temperature, Journal of Chemical Physics 72 (1980) 2384–2393.
[86] M. Parrinello, A. Rahman, Polymorphic transitions in single crystals: a new molecular dynamics method, Journal of Applied Physics 52 (1981) 7182.
[87] P.J. Hoogerbrugge, J.M.V.A. Koelman, Simulating microscopic hydrodynamics phenomena with dissipative particle dynamics, Europhysics Letters 19 (1992) 155–160.
[88] P.P. Ewald, Die berechnung optischer und elektrostatischer gitterpotentiale, Annalen der Physik 369 (3) (1921) 253–287.
[89] R. Hockney, J. Eastwood, Computer Simulation Using Particles, Taylor & Francis Group, New York, 1988.
[90] T. Darden, L. Perera, L. Li, L. Pedersen, New tricks for modelers from the crystallography toolkit: the particle mesh Ewald algorithm and its use in nucleic acid simulations, Structure 7 (3) (1999) R55–R60.
[91] J.D. Halverson, W. Lee, G.S. Grest, A.Y. Grosberg, K. Kremer, Molecular dynamics simulation study of nonconcatenated ring polymers in a melt: I. statics, Journal of Chemical Physics 134 (2011) 204904.