*n*Moldyn 3: Using Task Farming for a Parallel Spectroscopy-Oriented Analysis of Molecular Dynamics Simulations

Konrad Hinsen,[a,b] Eric Pellegrini,[c] Sławomir Stachura,[a,b] and Gerald R. Kneller*[a,b,d]

We present a new version of the program package *n*Moldyn, which has been originally developed for a neutron-scattering oriented analysis of molecular dynamics simulations of macromolecular systems (Kneller et al., Comput. Phys. Commun. 1995, 91, 191) and was later rewritten to include in-depth time series analyses and a graphical user interface (Rog et al., J. Comput. Chem. 2003, 24, 657). The main improvement in this new version and the focus of this article are the parallelization of all the analysis algorithms for use on multicore desktop computers as well as distributed-memory computing clusters. The parallelization is based on a task farming approach which maintains a simple program structure permitting easy modification and extension of the code to integrate new analysis methods. © 2012 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23035

Introduction

With the advent of computers, their usefulness to perform "virtual experiments" on many particle systems was rapidly recognized. Monte Carlo and molecular dynamics (MD) simulations[1,2] established themselves very soon as important tools for the development of the theory of liquids and dense gases. With increasing computing power, more complex molecular systems, such as molecular liquids and crystals, polymers, and biological macromolecules could be simulated and corresponding simulation programs and force fields were developed. We mention here the CHARMM,[3] AMBER,[4] and GROMOS[5] packages, which were among the first codes allowing to perform MD simulations of biomolecular systems and molecular systems in general, and which are each associated with a force field of the same name.

To fully exploit the information content in MD trajectories of complex molecular systems, advanced analysis tools are needed. Many analysis algorithms are in fact comparable to MD simulation algorithms in terms of complexity and runtime. In addition, their I/O overhead is often considerable and must be taken into account in algorithm design and implementation, as well as in the choice of file formats and data management techniques. The program package *n*Moldyn has been developed to make state-of-the-art analysis algorithms available to both experimentalists and computational scientists and thus to facilitate the confrontation of theory and experiment that is at the core of scientific research. Originally designed for the needs of the neutron scattering community, it has been extended progressively to cover other spectroscopical techniques.

The first version of *n*Moldyn was published more than 15 years ago[6] and introduced three major technological advances: (1) the use of fast Fourier transforms to accelerate the computation of time correlation functions, a technique that was well established in engineering[7] but completely ignored in the molecular simulation community, (2) quaternion-based analysis of rigid-body motions of molecules or parts thereof, and (3) a specially designed file format enabling fast access to the MD trajectory data for all the access patterns required by the analysis algorithms.

The second version of *n*Moldyn[8] was a complete rewrite whose goal was a more modular and more maintainable program that permitted easier integration of new analysis algorithms. A graphical user interface was added in reply to numerous requests from a growing user base. To this end, *n*Moldyn was rewritten in the object-oriented programming language Python,[9] profiting from the large ecosystem of scientific libraries available for that language. The most important basis for *n*Moldyn development since version 2 has been the Molecular Modeling Toolkit (MMTK),[10] which supplies the in-memory representation of molecular systems as well as access to trajectories in the netCDF format.[11] Other important support libraries are Numerical Python,[12] Scientific Python,[13] and FFTW.[14]

In addition to the technical improvements, *n*Moldyn 2 also provided enhanced functionality: (1) maximum entropy estimation of time correlations and their Fourier spectra,[15,16] and (2) the computation of memory functions,[17] which play an

[a] K. Hinsen, S. Stachura, G. R. Kneller
Centre de Biophysique Moléculaire (CNRS), Rue Charles Sadron, 45071 Orléans, France
E-mail: gerald.kneller@cnrs-orleans.fr

[b] K. Hinsen, S. Stachura, G. R. Kneller
Synchrotron SOLEIL, Division Expériences, L'Orme des Merisiers, BP 48, 91192 Gif-sur-Yvette, France

[c] E. Pellegrini
Computing for Science group, Institut Laue-Langevin; 6 rue Jules Horowitz, BP 156, 38042 Grenoble, France

[d] G. R. Kneller
Université d'Orléans, Château de la Source-Av. du Parc Floral, 45067 Orléans, France

Contract/grant sponsor: Agence Nationale de la Recherche; Contract/grant numbers: ANR-06-CIS6-012-01 and ANR-2010-COSI-001-01.

important role in the theory of relaxation processes and the dynamics of condensed matter systems in general.[18]

This article describes version 3 of *n*Moldyn, whose main new feature is the parallel execution of all its analysis algorithms. We also added tools for trajectory manipulation, in particular trajectory converters for various MD programs and a method for the subtraction of global motions from MD trajectories of macromolecules. Finally, the graphical user interface has been overhauled and extended, most notably by more elaborate plotting features for the calculated results.

Parallelization of the Analysis Algorithms

Some of the computations implemented in *n*Moldyn are very time-consuming, requiring an execution time comparable to that of a MD simulation. It is therefore highly desirable to be able to use modern parallel computers to speed them up. From an algorithmic point of view, the parallelization of most of the analyses is straightforward. However, there are additional criteria that need to be taken into account due to the way *n*Moldyn is used in practice.

A characteristic feature of the analysis of MD trajectories is that the computations are much less standardized than simulation algorithms. Although some scientists use *n*Moldyn as a "black box" program for performing common analyses, others use it as a convenient starting point for doing personalized analyses that involve modifying the *n*Moldyn source code. These scientists are typically not experienced programmers, much less parallel computing experts. It is therefore important that the source code for the parallelized analysis algorithms remains understandable and modifiable for application scientists. The code must reflect the mathematical formulas that define the quantities to be calculated.

The modifiability criterion was already a main motivation in using a high-level language, Python, for the implementation of *n*Moldyn, trading in some computational efficiency for better understandability. For the same reason, parallelizing *n*Moldyn is not just a matter of choosing the most efficient implementation strategy. Again, we trade in some performance for simplicity of implementation, in particular by limiting ourselves to a few simple parallelization schemes whose relation to the mathematical definition of the calculated quantities is easy to follow.

Another criterion in our parallelization effort was to have a single Python code base for all computing platforms, ranging from single processor or multiprocessor desktop systems to distributed-memory clusters with multiprocessor nodes. Scientists implementing their own analyses on top of *n*Moldyn should be able to test their modifications on their desktop computers and then run the exact same code on a faster production platform. This again leads to less-than-optimal parallelization for a given platform, a price we are willing to pay for gaining flexibility.

Categories of algorithms

A close look at the computational analyses implemented in *n*Moldyn shows that there are three categories relevant for parallelization:

1. Independent calculations for multiple *q*-shells in neutron scattering intensities, such as for the dynamic structure factor.[8,19] Here, *q* denotes the modulus of the momentum transfer in a neutron scattering experiment, measured in units of $\hbar$.

2. Sums over the time steps in a trajectory, where the quantity being summed is an arbitrary function of the atomic positions. An example is the radial distribution function (RDF).

3. Sums over atoms or groups of atoms, where the quantity being summed is typically a time correlation function for a single-atom trajectory. An example is the velocity autocorrelation function (VACF), which is computed as the (normalized) sum of single-atom VACFs.

Because the calculation of the sums is very fast compared to the calculation of the terms to be added, all three categories represent what is sometimes called "embarrassing parallelism:" the total computation can be trivially decomposed into completely independent operations that can be assigned to different processors. Therefore, the parallelization of *n*Moldyn presents no difficulty at all from a purely algorithmic point of view.

Some analyses can be assigned to several categories. For example, the elastic incoherent structure factor is calculated as an average over time steps and as a sum over atoms, but also independently for each of the *q*-shells. In those cases, we chose the category that yields the fastest operation for typical input data sets. Users do not need to be aware of the different categories.

The fundamental approach to parallelism taken by *n*Moldyn is known as "task farming": a master process hands out subtasks to the available slave processes and then collects the results as they are sent back by the slaves, combining them to produce the final output file. Task farming is attractive because it is a particularly simple and moreover a "deterministic" parallelization strategy. The result of a task farming computation does not depend on the number, nature, or workload of the individual compute nodes, in contrast to more popular implementation techniques for parallelism, such as message passing, which require the application program to ensure determinism by judicious use of the message passing primitives. Task farming is one of the building blocks of algorithmic skeletons,[20] an approach to parallel computing that attempts to identify standard patterns that occur frequently in parallel algorithms.

The main drawback of task farming is that it is less flexible than other approaches. For *n*Moldyn, the main restriction is that no further parallelization is possible inside a subtask. For today's most common problem sizes and computer equipment, this is not a problem. The number of time steps or atoms in any trajectory big enough to make parallel computing worth the effort is much larger than the number of processors a user can typically dedicate to an *n*Moldyn run. Parallelization by *q*-shells is the most critical scheme, because the number of *q*-shells is often of the same order of magnitude (20–100) as the number of available processors on a cluster. It can thus happen that scaling is limited by the number of subtasks available for distribution and an additional small factor could be gained with a more flexible parallelization scheme if significant computing resources are available. However, the price to pay in terms of code complexity seems too high for today's dominant usage patterns.

Parallel computing architectures

A significant technical difficulty is the diversity of parallel computing architectures currently in use for scientific computing. Nearly all desktop and portable computers produced today use multicore processors and are thus shared-memory parallel machines. They are typically used by a single person who usually wants to use all available resources for performing a computation. Local compute servers, personal or shared among a research group, have the same basic architecture, but a larger number of cores and a larger number of users that have to respect resource allocation limits. Clusters are distributed-memory machines that vary widely in terms of per-node processing power, memory, disk storage, and network performance. They may or may not have shared disk storage for large data items such as MD trajectories, and access to them may or may not be sufficiently fast for I/O-intensive programs such as *n*Moldyn. Finally, supercomputers tend to have significant central processing unit (CPU) power and good network performance, but vary widely in I/O performance and structure.

The challenge for programs like *n*Moldyn is to exploit all of the architectures listed above efficiently while at the same time remaining straightforward to use on the most common platforms. A particularly important aspect is the handling of I/O. The task farming approach limits output operations to the master process. This is not a problem for *n*Moldyn because output data files are relatively small. Reading the typically big trajectory files, however, is time critical. As this data is constant during the computation, it is possible to have each node read the data independently from a file without violating the rules of the task farming approach. It is even possible to make multiple copies of the trajectory file and have each node read a different copy.

Parallelization framework

The popularity of the Python language for scientific computing and the importance of parallelization in this domain have led to the development of a large number of parallelization frameworks for Python, including several that propose straightforward implementations of the task farming approach. The most popular one is probably the multiprocessing module that is part of the Python standard library. The main criteria for choosing a parallelization framework for *n*Moldyn were (1) support for shared-memory and distributed-memory machines, (2) the possibility of handling trajectory access efficiently, and (3) flexibility and ease of use on the most popular parallel computing platforms, which are multicore desktop computers and distributed-memory computing clusters. These criteria led to the choice of the parallel computing framework that is implemented as part of the ScientificPython package[13] in the module Scientific.DistributedComputing.

Programs based on this framework define a master process that defines subtasks and retrieves their results using framework functions. In the case of *n*Moldyn, a subtask corresponds to calculating the VACF for a single atom, or a RDF for a single time step of the trajectory. A client program must also provide implementations for all its tasks in the form of a class containing corresponding methods, or in the form of a module containing corresponding functions. Finally, a client program can define shared read-only data objects and initialization code run by slave processes, a feature used by *n*Moldyn to open and preprocess trajectory files once per slave process, which reduces the per-task execution time.

The parallelization framework in ScientificPython provides a task manager that offers significant flexibility to the user. Each computational task (e.g., a single analysis run in *n*Moldyn) identifies itself with a unique name. The task manager permits the user to monitor the progress of this task by showing the number of active processes and the machines they run on, as well as the number of waiting and running subtasks. The task manager also provides an option to add another slave process to a task, on any computer that has a network connection to the computer running the master process. Finally, the task manager monitors the activities of each slave process, watching out in particular for slave processes that disappear. In that case, the unfinished subtasks that were run by that slave process are reassigned to other slaves. This system makes it possible to use all available machine resources at any given instant, including resources that are not guaranteed to be available over the whole duration of the computation. It it thus possible to start an *n*Moldyn job with a small number of slaves and add more slaves during the night or over a weekend, when more computational power is available.

For communication between the processes, the ScientificPython parallelization framework uses the Python Remote Object (Pyro) package.[21] Pyro is a remote-object broker similar in spirit to CORBA,[22] but limited to a single programming language (Python) and therefore much simpler. Pyro is used to publish a central task manager object that communicates with the master process and all slave processes to coordinate the computation.

Running parallel analyses

From the user's point of view, running an *n*Moldyn analysis in parallel is particularly easy on a shared-memory multiprocessor computer, a category that includes virtually all desktop and portable computers produced today. On such a machine, it is sufficient to select "multiprocessor" mode instead of "monoprocessor" mode and to choose the number of processors to be used by *n*Moldyn.

Parallel execution on a cluster or a supercomputer is more complicated because of the large diversity in I/O handling and job management between different systems. It is important to understand how file access and slave jobs are handled by *n*Moldyn to choose the best approach for a specific machine.

On a typical cluster or supercomputer, resources are attributed to jobs by a job scheduler such as the popular Portable

Batch System (PBS). At the beginning of a job, the job scheduler runs a shell script submitted by the users and passes the list of reserved nodes to this shell script through an environment variable. This mode of operation requires that all computations are fully specified in the shell script, excluding the use of interactive programs such as *n*Moldyn's graphical user interface. *n*Moldyn, therefore, allows the user to save an analysis as a Python script instead of running it directly. This Python script can then be run from batch job's main shell script.

For running such an analysis in parallel, the user needs to specify "cluster" mode instead of "monoprocessor" mode when generating the Python script. In cluster mode, the Python script contains only the master process that defines the subtasks and combines their results into the final output file. The slave processes that perform the actual computations must be started separately using the task manager provided by the parallelization framework in ScientificPython. A typical shell script describing an *n*Moldyn job first starts the master process and then the desired number of slaves, distributing them over the available nodes.

Trajectory files are opened using the same full path on all compute nodes. This leaves two choices to the user of a cluster or supercomputer: (1) put the trajectory files on a shared disk accessible from all nodes or (2) place a copy of the file on a local disk on each node that is accessible by the same path everywhere (typically /tmp or /scratch). The second approach is preferable on clusters on which local disk access is significantly faster than network-mediated shared disk access. It is the user's responsibility to make the file copies at the beginning of a batch job before running *n*Moldyn.

Parallel performance

To evaluate the performance of *n*Moldyn on parallel machines, we performed a series of benchmarks. We used a trajectory for a system consisting of a fully hydrated 1-palmitoyl-2-oleoyl-sn-glycero-3-phosphocholine (POPC) lipid bilayer, consisting of 68,129 atoms in total (36,716 lipid atoms and 31,413 solvent atoms). The trajectory contained 10,001 frames of a MD simulation covering a simulation time of 150 ns. The trajectory was produced using the GROMACS simulation software[23] and then converted to MMTK's netCDF-based trajectory format used by *n*Moldyn.

The benchmarks were run on Synchrotron SOLEIL's computing cluster, whose nodes each have two Intel quad-core processors running at 2.8 GHz and 36 GB of memory. The trajectory file used as the input for all analyses was stored on an network file system (NFS) hosted by a dedicated I/O node. The cluster's compute nodes are only accessible through batch jobs managed by the PBS. Each benchmark run was designed as a PBS job script that first starts the master process by running the Python script generated by *n*Moldyn. This master script defines the subtasks to be handled by slave processes, but does not do any computation itself. The job script then waits a few seconds, records the wall-clock time, and immediately launches slave jobs on the nodes attributed by PBS. Once all slave processes have ended, the wall-clock time is recorded again. The timings we report are the differences between the two wall-clock readings. They include all of the computation time and almost all time spent in communication and administrative overhead. Only the startup time of the master process is excluded. The reason for this choice is the unpredictable initialization time of the Pyro-based communication system, which requires the few-second wait between starting the master process and starting the slave processes. This delay is of no consequence in normal use (the initialization time of a PBS job is much longer anyway), but would add an undesirable near-random element to our benchmarking procedure.

We have run three different analyses to evaluate the three parallelization categories: (1) the coherent dynamic structure factor (CDSF), [8,19] calculated independently in parallel for each of 50 *q*-shells each containing 50 *q*-vectors, (2) the RDF,[8,24] calculated as a parallelized average over pair distance histograms for each time frame in the trajectory, and (3) the mean square displacement (MSD),[8,24] calculated as a parallelized sum over single-atom MSDs. The calculation of the RDF is somewhat artificial because its physical interpretation relies on the isotropy of the system, making it an uninteresting quantity for our essentially two-dimensional membrane system. However, this makes no difference for the purpose of performance measurements. The execution time for our three benchmarks are shown in Figures 1–3. The insets in these figures show the scaling behavior, calculated as the inverse of the execution time normalized to the execution time on a single processor. All computations were run on eight-core nodes, meaning that the runs for 1 and 4 cores do not fully exploit the allocated computing resources. The two data sets for the MSD calculation correspond to different file layouts, as will be discussed later.

Comparing the three figures, we note near-perfect scaling for the RDF, good scaling for the CDSF, and rather bad scaling for the MSD. We attribute this difference essentially to the differences in I/O overhead between the three analyses. The near-perfect scaling in Figure 2 corresponds to a combination of efficient data access (the data for one time frame is stored

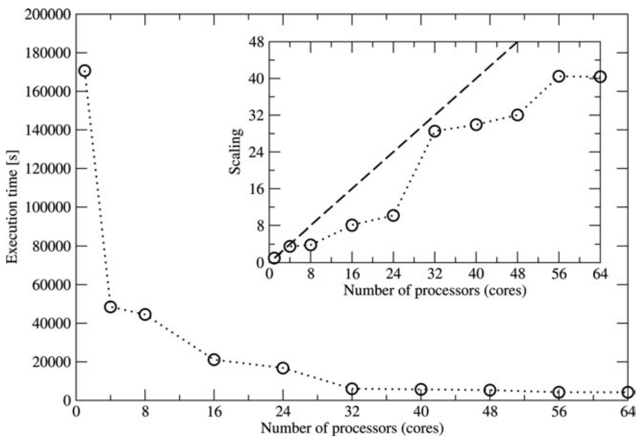

Figure 1. Execution times and scaling as a function of the number of processors for a CDSF computation. The parallelization is done by distributing the 50 *q* shells over the processors. This explains why the execution time does not go down when moving from 56 processors to 64 processors: the additional processors remain idle.

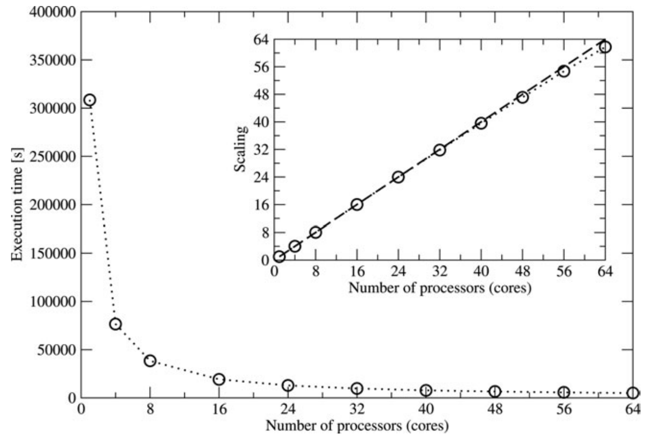

Figure 2. Execution times and scaling as a function of the number of processors for a RDF computation. The parallelization is done by distributing the 10,001 time frames over the processors.

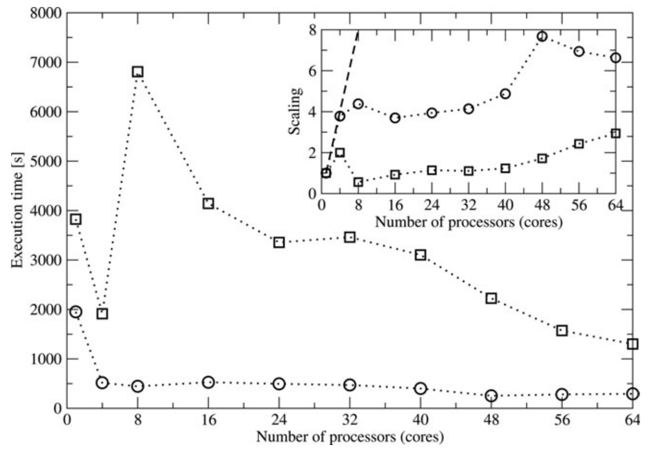

Figure 3. Execution times and scaling as a function of the number of processors for a MSD computation. The parallelization is done by distributing the 68,129 atoms over the processors. The bad scaling is due to the important I/O overhead of the MSD computation. The circles are for a trajectory with a block size of 500, the squares are for a block size of 1 (see text for an explanation).

contiguously in a trajectory file) and a large amount of CPU time for processing this data. At the other extreme, the bad scaling in Figure 3 is the result of an unfavorable data access pattern (the whole trajectory file must be traversed for reading one atom's data) and an inexpensive computation. In between (Fig. 1), we see a moderate CPU effort and the need to read the trajectory once per *q*-shell, that is, 50 times.

To verify that the scaling behavior of the MSD computation is related to the I/O overhead, we compared the execution times for two different trajectory file layouts (see Fig. 3). The parameter that is varied is the "block size" parameter given to MMTK when creating a trajectory file. It specifies the number of time steps in one trajectory block. Inside a trajectory block, the data are arranged such that the atom index varies faster than the time step index. In other words, the coordinates for one atom are stored contiguously for the time steps inside one block. A block size of 1 means storage by time frames, which is the layout used in all other common file formats for MD simulations. It is optimal for reading or writing frame by frame, but unfavorable for access by atom. Conversely, a block size equal to the number of time steps is optimal for access by atom, but unfavorable for access by time frame. A block size of 500 is a reasonable compromise for the trajectory we use for our benchmarks. The circles in Figure 3 correspond to this choice. It is much faster for the MSD computation than the default block size of 1 (timings shown as squares), in particular when more than one node is used to process the file. The jump in execution time when moving from four to eight nodes is presumably caused by increasing competition between the processors for access to the remote disk, but an in-depth analysis of this phenomenon would require more information about the I/O subsystem of our cluster than we have at our disposal.

Parallel I/O

Unfortunately, parallel I/O has received much less attention from researchers and computer manufacturers than parallel computation. It is still difficult if not impossible to write portably efficient parallel software if I/O costs are significant. In the current version of *n*Moldyn, we have not attempted to optimize parallel I/O in any way, relying fully on the computer's operating system and the user's understanding of I/O efficiency on a given system. Providing better parallel I/O performance is one of our goals for a future *n*Moldyn release. This will be another major effort because it requires building an additional abstraction layer that separates the scientific algorithms from platform-specific optimized I/O handling.

To illustrate the strong dependence of I/O optimizations on the characteristics on the hardware and systems software, we will look at a possible modification that at first glance looks like an obvious improvement. In our "parallelization by *q* shell" scheme, every subtask (doing the computation for one *q* value) reads exactly the same information from the same file, and moreover at approximately the same time. This creates a high load on the I/O system of the parallel computer. It seems preferable to have a single processor load the data from the input file and distribute it to the other processors. However, there is one configuration where this would result in decreased performance: a cluster whose nodes have local disks to which the input file has been copied before starting the computation. Clusters with node-local disks are common, and copying the input file is easy, which means that this configuration is practically relevant. This example shows that I/O optimization requires some part of the software to be optimized for each specific hardware configuration.

Other New Features in nMoldyn 3

The graphical user interface of *n*Moldyn 3 has been completely rewritten. As before, it can run analyses directly or produce a script for running the analysis on a different machine or through a batch system. Both ways of running analyses now support parallel execution. In addition, the new graphical user interface provides a sophisticated interactive plotting facility for the results produced by the analyses, including three-dimensional plotting for functions of q and t (intermediate scattering functions) or q and ω (dynamic structure factors).

*n*Moldyn 3 also provides additional trajectory converters that create MMTK-format netCDF trajectory files from various input formats. *n*Moldyn 3 can convert trajectories generated with the simulation programs Amber, CHARMM, DL_POLY, Materials Studio, NAMD, VASP, and X-PLOR.

Conclusions

The new release of *n*Moldyn described here provides significant performance improvements through parallelization. For the most time-consuming analyses, it offers good scaling on typical computing clusters. The flexible parallelization technology makes it possible to add and remove computing nodes while a computation is running and thus adapt resource usage to resource availability. *n*Moldyn 3 is freely available for download.[25]

Keywords: parallel computing · task farming · time series analysis · theoretical spectroscopy · molecular simulation

How to cite this article: K. Hinsen, E. Pellegrini, S. Stachura, G. R. Kneller, *J. Comput. Chem.* **2012,** *33,* 2043–2048. DOI: 10.1002/jcc.23035

[1] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, *J. Chem. Phys.* **1953**, *21,* 1087.
[2] A. Rahman, *Phys. Rev.* **1964**, *136,* 405.
[3] B. R. Brooks, C. L. Brooks, III, A. D. Mackerell, Jr., L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, 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, *J. Comp. Chem.* **2009**, *30,* 1545.
[4] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, *J. Comput. Chem.* **2005**, *26,* 1668.
[5] W. R. P. Scott, P. H. Hnenberger, I. G. Tironi, A. E. Mark, S. R. Billeter, J. Fennen, A. E. Torda, T. Huber, P. Krger, W. F. van Gunsteren, *J. Phys. Chem. A* **1999**, *103,* 3596.
[6] G. Kneller, V. Keiner, M. Kneller, M. Schiller, *Comput. Phys. Commun.* **1995**, *91,* 191.
[7] E. Brigham, The Fast Fourier Transform; Prentice Hall: Englewood Cliffs, NJ, 1974.
[8] T. Rog, K. Murzyn, K. Hinsen, G. Kneller, *J. Comput. Chem.* **2003**, *24,* 657.
[9] G. van Rossum, The Python web site http://www.python.org/. Accessed on June 1, 2012.
[10] K. Hinsen, *J. Comp. Chem.* **2000**, *21,* 79.
[11] Unidata, Network common data form (netcdf). http://www.unidata.ucar.edu/software/netcdf/.
[12] D. Ascher, P. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, Numerical Python Technical Report UCRL-MA-128569; Lawrence Livermoore National Laboratory: Livermore, CA, **2001**.
[13] K. Hinsen, ScientificPython web site http://www.dirac.cnrs-orleans.fr/ScientificPython/. Accessed on June 1, 2012.
[14] The FFTW library http://fftw.org/. Accessed on June 1, 2012.
[15] A. Papoulis, Signal Analysis; McGraw Hill: New York, **1984**.
[16] A. Papoulis, Probability, Random Variables, and Stochastic Processes, 3rd ed.; McGraw Hill: New York, **1991**.
[17] G. Kneller, K. Hinsen, *J. Chem. Phys.* **2001**, *115,* 11097.
[18] R. Zwanzig, Nonequilibrium Statistical Mechanics, Oxford University Press USA: New York, **2001**.
[19] S. Lovesey, Theory of Neutron Scattering from Condensed Matter, Vol. I; Clarendon Press: Oxford, **1984**.
[20] M. Cole, Algorithmic Skeletons: Structural Management of Parallel Computation; MIT Press: Cambridge, MA, **1989**.
[21] I. de Jong, Pyro web site http://www.xs4all.nl/~irmen/pyro3/. Accessed on June 1, 2012.
[22] The Object Management Group, Common Object Request Broker Architecture. http://www.corba.org/.
[23] D. V. D. Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, H. J. C. Berendsen, *J. Comput. Chem.* **2005**, *26,* 1701.
[24] J. Boon, S. Yip, Molecular Hydrodynamics; McGraw Hill: New York, 1980.
[25] nMOLDYN web site http://dirac.cnrs-orleans.fr/nMOLDYN/. Accessed on June 1, 2012.

Received: 13 April 2012
Revised: 11 May 2012
Accepted: 19 May 2012
Published online on 8 June 2012