Software design for a highly parallel molecular dynamics simulation framework in chemical engineering[☆]

M. Buchholz[a,*], H.-J. Bungartz[a], J. Vrabec[b]

[a] *Technische Universität München, Institut für Informatik*
[b] *University of Paderborn, Thermodynamics and Energy Technology*

Abstract

The software structure of MarDyn, a molecular dynamics simulation program for nanofluidics in chemical engineering, is presented. Multi-component mixtures in heterogeneous states with huge numbers of particles put great challenges on the simulation of scenarios in this field, which cannot be tackled with the established molecular simulation programs. The need to develop a new software for such simulations with an interdisciplinary team opened the chance of using state-of-the-art methods on the modelling as well as on the simulation side. This entails the need to test and compare different methods in all parts of the program to be able to find the best method for each task. It is shown how the software design of MarDyn supports testing and comparing of various methods in all parts of the program. The focus lies on those parts concerning parallelisation, which is on the one hand a pure MPI parallelisation and on the other hand a hybrid approach using MPI in combination with a memory-coupled parallelisation. For the latter, MarDyn not only allows the use of different algorithms, but also supports the use of different libraries such as OpenMP and TBB.

Keywords: molecular dynamics, hybrid parallelisation, load balancing, numerical software

---

[☆]This work was supported by the Bundesministerium für Bildung und Forschung within the project "IMEMO—Innovative HPC-Methoden und Einsatz für hoch skalierbare Molekulare Simulation", Förderkennzeichen 01IH08013D.

[*]Corresponding author

1. Introduction

Molecular dynamics simulations are widely used especially in biochemistry and solid-state physics. Their use in chemical engineering applications is less prevalent. Most simulations in this field are mainly restricted to continuum methods such as computational fluid dynamics or, on even more aggregated scales, by means of mass and energy balances. There are several reasons which recently advanced the use of molecular methods in this field. On the modelling side, scientific progress has brought up more realistic models based on force fields representing intermolecular interactions for many industrially relevant fluids. This allows the study of effects, such as nucleation processes for complex mixtures, e.g., on a molecular scale, which were before not accessible by such simulations. Those scenarios, covering nanoscale processes, use huge numbers of particles and require modern HPC systems with a large number of processing units. The heterogeneous and dynamically changing distribution of particles necessitates complex load balancing algorithms. The hierarchical layout of new HPC systems is another challenge that favours the use of hybrid parallelisation concepts. At the same time, the complexity of the underlying models for the simulation is also very high and further increasing, as a large variety of materials at different states has to be considered, e.g. when the flow of a fluid with heterogeneous density in a solid channel is simulated. To develop software for industrially relevant applications in the field of chemical engineering, experts from different disciplines have to work together. This entails high demands on the structure of the simulation software.

In this paper, the software structure of MarDyn is presented, a simulation program for nanofluidics written in C++, which was developed to tackle, amongst others, these challenges. We start with a glimpse on the peculiarities of the target applications in chemical engineering, followed by a presentation of the software structure. Then, the focus is put on those parts relevant for memory- and message-coupled parallelisation and load balancing, and it is shown how the software design eases the implementation and comparison of different parallelisation concepts. Finally, runtime results for the different parallelisation strategies are presented. Part of this work was done within the project IMEMO with the collaboration of the Laboratory of Engineering Thermodynamics at the University of Kaiserslautern, the chair for Thermodynamics and Energy Technology at the University of Paderborn, the High Performance Computing Center Stuttgart, the Fraunhofer Institute for Industrial Mathematics , and the Chair for Scientific Computing at the Technische Universität München.

2. Molecular dynamics in chemical engineering

Molecular simulations are usually carried out in the following fashion: First, a molecular model has to be chosen, then it is assigned to a molecular configuration, and finally, the phase space is explored for a given number of iteration steps under specified boundary conditions, which allows gathering a broad variety of information. The physically most important input are molecular models (also known as force fields) which describe the interactions between atoms or molecules in terms of parameterised potential functions. The intermolecular forces govern the simulation result due to the fact that the “external” forces due to the boundary conditions, such as thermostats or velocity gradients, are usually much weaker.

Phase space can be sampled either stochastically by Monte Carlo methods or deterministically by molecular dynamics, where the latter allows a straightforward access to time-dependent properties. Molecular dynamics is based on the assumption that molecules are bodies from classical mechanics, i.e. that they propagate through space following Newton’s equations of motion. These are a set of second order differential equations that can be solved numerically via time-integration.

The mathematical properties of the molecular models strongly influence data structures and algorithms and thus determine the design of simulation software. Here, the focus lies on short-ranged potential functions, which are well suited to describe non-polar fluids such as hydrocarbons that have a central role in chemical engineering. The most widespread example for such a potential function is the Lennard-Jones$^{12-6}$ (LJ) potential [1, 2]

$$U_{\mathrm{LJ}}\left(r_{ij}\right) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) , \tag{1}$$

where r_{ij} is the distance between two LJ sites, i.e. atoms or molecular groups. The LJ potential covers the two basic intermolecular interactions: repulsion at short distances due to overlap of electronic orbitals and dispersive attraction at intermediate distances due to mutual polarisation. The size parameter σ is a characteristic interaction distance, corresponding to a collision diameter, and the energy parameter ϵ is the magnitude of the dispersive interaction. Apart from the LJ-Potential, MarDyn currently supports dipoles, quadrupoles and the Tersoff-Potential, which is used to model hybridization states of carbon, e.g.

The intramolecular potential and hence also the force, which is the negative gradient of the potential, decrease rapidly with increasing distance, cf. (1). Therefore, for a given molecule, it is sufficient for the force calculation to only examine its local neighbourhood within a cutoff radius r_c for interacting molecules. This is of central importance for algorithms and parallelisations, as it allows the use of the Linked-Cells algorithm [3] to find neighbouring molecules that permits highly sophisticated parallelisation methods based on a spatial decomposition [4] of the simulation domain. To compensate for the cutoff part of the potential, cutoff corrections are calculated assuming a homogeneous particle distribution outside of the cutoff radius.

MarDyn can be used for different scenarios in the field of chemical engineering. It is designed for spatially large systems, where the cutoff radius is small compared to the system size. The application that was used for the runtime results discussed below is homogeneous nucleation. Nucleation is a process where liquid droplets emerge from a supersaturated vapour (see Figure 1), e.g. when fog arises. Real experiments on nucleation are difficult and often yield inconclusive data so that supporting simulation data is of high interest [5]. The large number of molecules and simulation time steps needed for meaningful results is only feasible by the use of current HPC systems. Due to the strong heterogeneity of the molecule distribution in nucleation processes, great challenges are posed on the parallelisation and especially the load balancing of the simulation.

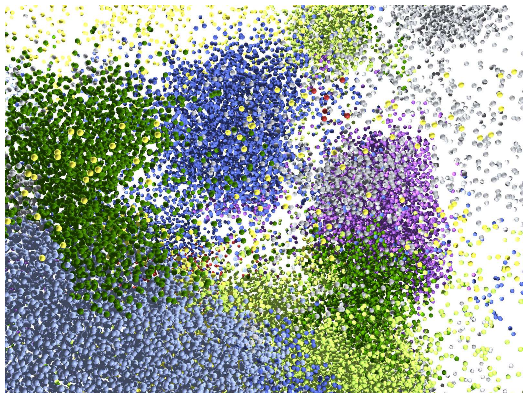

Figure 1: Detail of a visualisation of a nucleation simulation.

3. Software structure

In the introduction, the need for a well structured and modular software was emphasized. Only with a consistent partition of the simulation program into clearly specified tasks, it can be ensured that not every developer has to understand all parts of the software, which eventually gets impossible in an evolving software. There are many other reasons for a strictly modular software design, out of which one is especially important for scientific software. Quite often, the goal is not only to implement a certain functionality into a simulation code, but to compare different methods to achieve this functionality. This could be the comparison of different molecular models with respect to their accuracy or of algorithms or parallelisations concerning their efficiency. To obtain credible results from such a comparison, it has to be ensured that switching between different methods can be done easily and without any side-effects on the rest of the program. Another important argument for well structured software is performance. To be able to optimise the runtime of a program, data structures, algorithms and pieces of code in all parts of the software have to be tuned. This is only possible in a reasonable amount of time if all functionalities are clearly separated using a modular software design.

Figure 2 shows a very simplified class diagram of MarDyn. The boxes with grey italic labels represent interfaces, all others are classes. The central class of the simulation, which contains the main loop iteration over the time steps, is always associated with the interfaces and not with some concrete implementation. This allows a simple exchange of algorithms, e.g. for finding particle pairs [6] or integrating the equations of motion, without any effects on the other parts of the program.

The details and the advantages of this approach are described for two selected parts of the program which are particularly important for HPC. The first part contains the interface and different implementations for the message-coupled parallelisation. The software structure of that part is exemplary for most other parts of the program. The structure of the second part, the memory-coupled parallelisation, differs from the other parts and is therefore discussed separately. A prototypical GPU based parallelisation, which currently supports only the Lennard-Jones model, was also integrated in MarDyn and works together with the MPI parallelisation without any changes to the latter one. There won't be any details on the GPU parallelisation here, as work on this part is not finished yet, but first results for

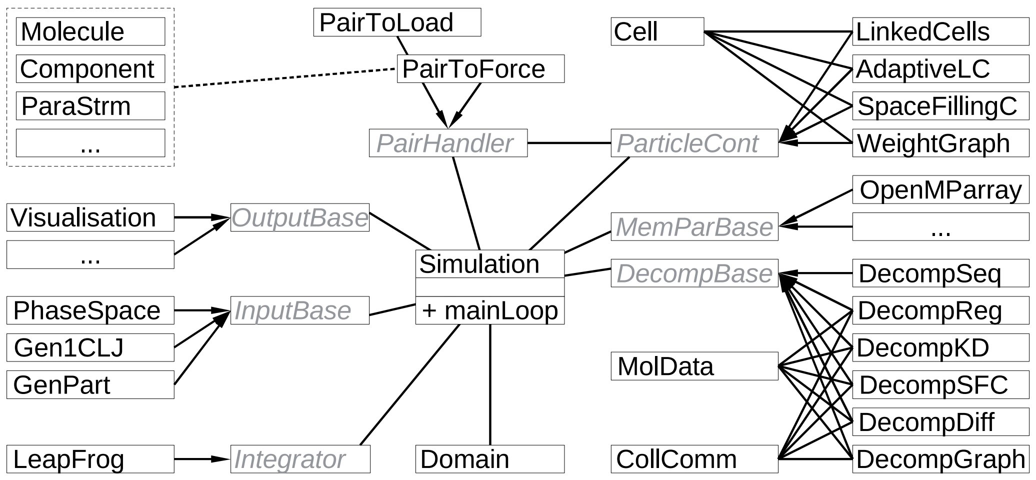

Figure 2: Class diagram of MarDyn.

Lennard-Jones-fluids are promising.

4. Parallelisation

When designing an interface for some functionality of a software, it is important to find the appropriate level of flexibility. It should allow for every imaginable implementation of the specified functionality. But this flexibility should neither have big drawbacks on the efficiency, nor reduce comprehensibility of the interface. Therefore, it is not feasible to design an interface which allows the implementation of arbitrary parallelisation algorithms. Therefore, the code is restricted to message coupling as parallelisation paradigm, as this is best suited for the given circumstances. Memory coupling can be used additionally, as shown later in this paper, but the basic parallelisation is based on message coupling. About twenty years ago, parallelisation started to become a relevant topic for molecular dynamics simulations. Different parallelisation approaches, which can be grouped into three classes, have emerged since then. Atom decompositions are based on a replication of the data on all processors, force decompositions distribute the work based on a partitioning of the force matrix. None of these methods is effective for very large numbers of particles, as memory consumption is too big, or for a large number of processors, as the number of communication partners is too large. So at least for the given applications, spatial decomposition approaches, which are based on a decomposition of the physical domain, are superior [7, 8]. This reduces the necessary flexibility of the interface to allow arbitrary message coupled spatial decomposition algorithms for the parallelisation. Based on the current paradigm of the application area for the software, the definition of such an interface – not all details will be discussed here – is sufficient to allow for the implementation of almost any useful parallelisation algorithm.

There is some functionality which logically belongs to the parallelisation, such as the exchange of molecules between the processes and collective communication commands. Having in mind that several researchers from different disciplines work on the same code, the different parts should be designed in a way that researchers working on one part do not interfere with and possibly do not even have to understand the other parts. But sending a molecule means that one has to know something about the molecule structure and the values needed to describe a molecule, as MPI does not allow the communication of arbitrary classes. When the molecular model changes, this might change the data to be sent; equally so for collective communications. Here, such operations are used for calculating thermodynamic properties, about which the parallelisation developer should not have to care.

After considering all arguments mentioned before, two new classes dealing with molecule data and data for collective communication were introduced. The class for the molecule data is relatively simple, all values needed to describe a molecule are packed together in a single MPI data type. The implementation of the collective communication in a separate class has to include yet another feature. At different stages during a simulation steps, different kinds of collective operations on a varying amount of values of any type can be performed. If several values have to be communicated, this should be done, as for the molecule data, with a single command. An exemplary workflow of how this is done with the class for collective communication is shown in the following algorithm:

```
CollectiveCommunication cc; // declare variable
cc.init(2);                 // two values to be sent
cc.appendInt(5);            // first value (int)
cc.appendDouble(3.72);      // second value (double)
cc.allReduceSum();          // Perform a global reduce command
int res1 = cc.getInt();     // retrieve first value (int)
int res2 = cc.getDouble();  // retrieve second value (double)
cc.finalize();              // free memory
```

Within the collective communication class, all appended values are packed together to one MPI data type, and an MPI operation is defined which can add variables of that type. Data type and operation can than be used to perform one of MPI's efficient built-in collective operations. Using the classes for collective communication and particle data, a concrete implementation of the parallelisation interface is strongly decoupled from all other parts of the simulation.

5. Results for load balanced parallelisation

One of the main goals of the modular approach using interfaces for all major parts was the possibility to exchange and compare different algorithms. As described in Section 2, particle distributions in nucleation processes are very heterogeneous. The density of liquid and gas differs by a factor of up to 1000 (in the extreme case of water at ambient conditions), which results in very different calculation costs. For the parallelisation five different algorithms have been developed [8], four of them implementing load balancing facilities:

- uniform decomposition: The physical domain is decomposed into a grid of cuboid regions. As no load balancing is done, this parallelisation is not suited for heterogeneous scenarios. But it is superior for homogeneous scenarios, because the number of communication neighbours is minimized (six per process).
- k-d-tree-based decomposition: The domain is recursively decomposed into cuboid sub-regions. In each step, the costs caused by each potential separation plane – communication costs and doubled computation on the created boundary – are also considered to choose the optimal decomposition.
- graph partitioning: For this parallelisation algorithm, the domain, which is divided into small cells, is mapped onto a graph. Calculation costs are mapped onto nodes and costs for potential boundaries are mapped onto edges. After partitioning the graph by using the library ParMetis [9], each partition is assigned to one process.
- space-filling curves: A space-filling curve is laid through the cells of the domain. Then the decomposition algorithm follows the curve and

cuts it into chunks such that each chunk represents an equal amount of the total load. The load is perfectly balanced, but it is not possible to optimise the course of the domain boundary.

- diffusion: Balanced load is achieved by a local exchange of cells between neighbouring processes.

As described in Section 3, those five methods are realised as an implementation of the same interface and are therefore easily exchangeable. In Fig. 3, the speedup using strong scaling is shown for all five parallelisations. The underlying scenario is a heterogeneous fluid with 15 liquid nuclei within a vapour phase and a total number of two million particles. The machine used for these and the following results is the NEC Nehalem cluster at the High Performance Computing Center Stuttgart. This machine has 700 nodes, each with two 2.8 GHz quad-core Intel Xeon Nehalem processors, connected over an Infiniband network. For single jobs, 2048 cores can used at the same time.

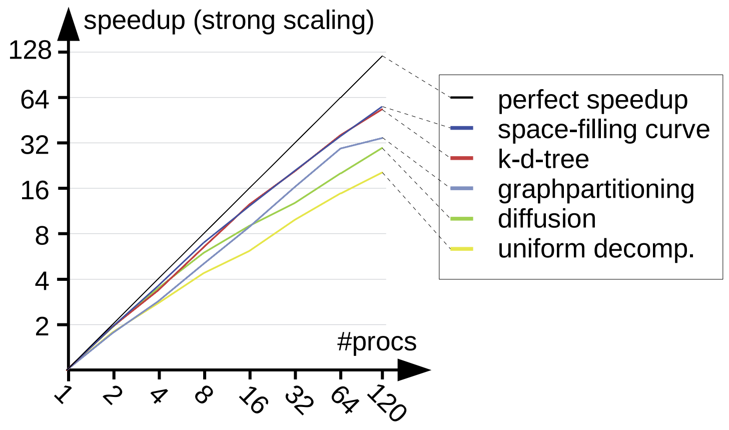

Figure 3: Strong speedup for a heterogeneous particle distribution with two million particles and a maximum of 120 processors.

The three methods which performed worst – besides the not load balanced parallelisation those based on graphs and diffusion – had problems, e.g., because they ran out of memory for large simulations. Therefore, the following results are only complete for the two best methods, using k-d-trees or space-filling curves, respectively. Figure 4 shows the speedup using weak scaling for a maximum of twenty million heterogeneously distributed particles on 2048 processors. All results were averaged over 100 time steps, load balancing was done once at the beginning of those 100 steps (twice for the methods using space-filling curves and diffusion, as they involve a correction step) and took less than one percent of the computation time.

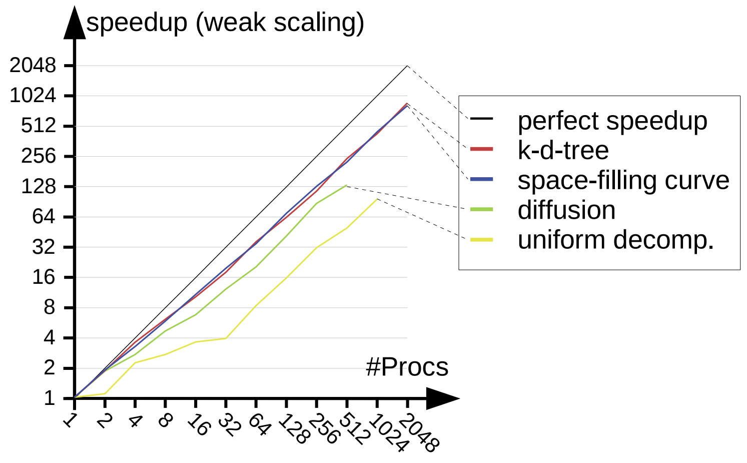

Figure 4: Weak speedup for a heterogeneous particle distribution with a maximum of twenty million particles on up to 2048 processors.

6. Memory-coupled/hybrid parallelisation

As for the MPI-based parallelisation, the memory-coupled parallelisation should also be separated from the rest of the program for the same reasons as discussed before. For the domain decomposition used in the MPI parallelisation, each process can basically work locally most of the time, only once per time step processes have to exchange data with their neighbours. The situation for the memory-coupled parallelisation is different, because the parallelisation is usually done for much smaller units or code pieces such as loops. OpenMP is the de-facto standard for memory-coupled parallelisation, but since multi-core processors have emerged, alternatives such as Intel's Thread Building Blocks (TBB) have become available. OpenMP uses preprocessor statements to do this parallelisation. One big disadvantage is therefore a source code full of preprocessor statements for many loops or other small pieces of code to be parallelised. TBB is an object-oriented library for C++, so its programming paradigm is completely different to that of OpenMP with its preprocessor statements. As none of the libraries is clearly superior to the other, MarDyn allows the interchangeable use of both libraries. Memory-coupled and message-coupled parallelisations can be used at the same time, which is called a hybrid parallelisation. In this case, shared-memory is used within each node of a HPC cluster (using all processors and cores available) and message coupling is used between the nodes.

As mentioned before, most parallelisation is done on a loop level. The single loop which dominates in terms of computation time is the one used to find all pairs of neighbouring molecules for the force calculation. Less expensive loops are those which do something for each molecule, such as the spatial molecule propagation or the thermostat loop. These can be parallelised straightforwardly, because there are no dependencies to be solved. The parallelisation of the force calculation is not that straightforward. Many different pairs share and therefore access the same molecule, which causes a race condition when two threads try to accumulate the calculated force for the same molecule. For an explanation why this happens, the Linked-Cells algorithm is shortly described. For this algorithm, the simulation domain is discretized into cells whose length equals the cutoff radius r_c of the short-ranged interaction potential. The core of the algorithm is a loop over all cells and for each of the cells a loop over all forward cells. For each cell pair, the interactions of all related molecules are calculated. Backward neighbours do not have to be considered as the force between two molecules is symmetric according to Newton's third law.

A simple loop parallelisation of the outer loop of the Linked-Cells algorithm assigns each thread a part of the loop. Two different threads can then be either directly neighbouring or have some common neighbouring cells, which can then cause the race condition. There are several different ways to get rid of this race condition, e.g. using atomic operations, thread-local intermediate storage for forces, blocking cell-access, or splitting the force loop into several loops, each with independent cells and therefore parallelisable. The details of these different methods will not be discussed, but each of the methods has advantages and disadvantages. Following the underlying software philosophy of MarDyn, it was important to allow for the implementation and comparison of all methods. For three of them, this was possible without big changes in the software framework. But the splitting of the force loop strongly interferes with the Linked-Cells algorithm and therefore has to be treated separately. As stated before, the conflict can arise in situations where two threads process either neighbouring cells or cells which share the same neighbouring cell. If the outer loop ran only over cells which are independent in such a way that they do not have common neighbouring cells, the race condition could not arise. For a two-dimensional example, such independent cells are marked in blue in Fig. 5 (left). So dividing the outer loop into six loops, each with independent cells, bypasses the race condition problem and, therefore, makes each of the loops parallelisable. In three dimensions, 18 loops are necessary for this decoupling.

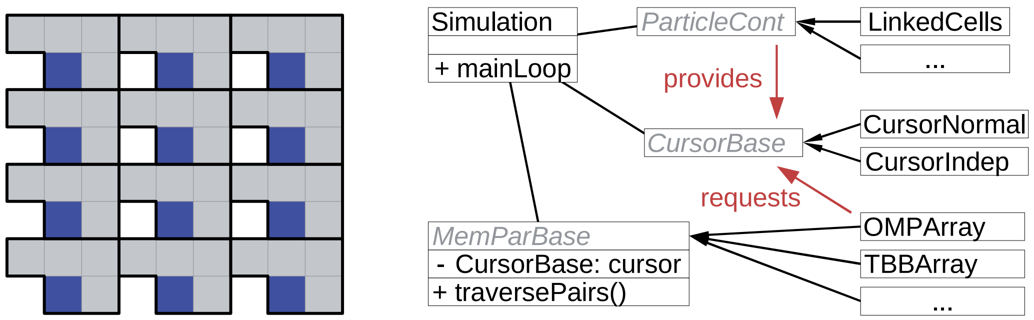

Figure 5: Independent cells for a two-dimensional example marked in blue (left); class diagram for the memory-coupled parallelisation in MarDyn (right)

The disadvantage of the described approach is that many small loops create more overhead, especially for the parallelisation, as one single big loop. This implicates that the single loop in the program should not be fully replaced, as multiple loops should only be used with a shared-memory or a hybrid parallelisation (shared-memory and distributed-memory parallelisations combined) – the program might run sequentially or with a pure MPI parallelisation – and if this specific strategy for avoiding race conditions in the force calculation is chosen. Therefore, a `Cursor`-interface has been added to MarDyn, as shown in Fig. 5 (right). Each implementation of this interface defines sets of cells which have to be traversed by the Linked-Cells algorithm. In the normal case, there is just one set containing all cells. For the described strategy, 18 independent sets are defined. A memory-coupled parallelisation (implementing `MemParBase`) can now choose which cursor should be used.

Some results of a hybrid simulation run for the different conflict-solving strategies are shown in Fig. 6. All of the strategies were implemented with OpenMP, three of them additionally with TBB (TBB does not support atomic operations directly, e.g.). For this simulation, and also for other typical nucleation scenarios, the hybrid solution cannot compare with the pure MPI parallelisation. The main reason for that is – apart from the fact that the MPI parallelisation is very good (see Section 5) – that there are some parts which are very hard to parallelise with the shared-memory approach. In the hybrid case, this is especially the MPI communication which is currently done by a single thread per node.

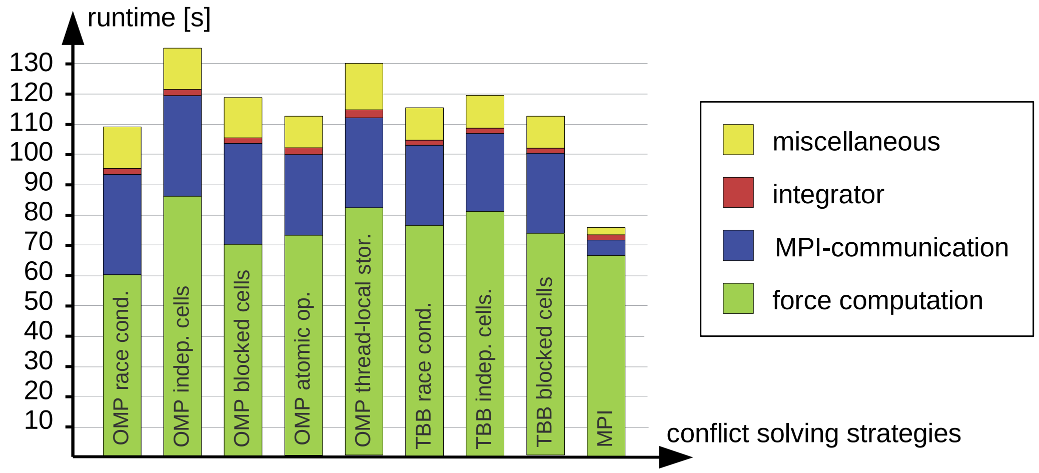

Figure 6: Hybrid runtime with eight cores per Nehalem node using different conflict solving strategies, broken down into time for force-calculation, MPI communication, integrator and miscellaneous. The runtime of a pure MPI parallelisation is also shown. To avoid false measurements caused by idle processes in heterogeneous scenarios, the measurements very taken on a single node.

7. Conclusion

Some features of the software design of MarDyn, a molecular dynamics program for chemical engineering applications, have been described. The main focus was on how to allow the use of different algorithms for solving problems in all parts of the simulation program in such a way that the algorithms can be compared and evaluated easily. Some more details and results were shown for those parts of the program dealing with parallelisation. For example, for the MPI parallelisation of the shown scenario, strategies based on k-d-trees and space-filling curves are best and show a comparable performance. Which one should be chosen depends on the specific scenario. For the simulation of a single big drop, e.g., the k-d-tree based load balancing is clearly superior to that based on space filling curves. And for homogeneous particle distributions, as they occur, e.g., in the beginning of a nucleation simulation before nucleation has started, the regular domain decomposition without load balancing is best. The possibility to choose between different parallelisation strategies is also important for possible future applications, which might involve point charges and therefore necessitate long-range interactions. Especially parallelisation strategies based on space-filling curves or trees are best suited to support such interactions, e.g. using fast multipole methods. Here, the software design of MarDyn allows the selection of that approach which is best for each specific scenario. The developed software design helps researchers working with the program to focus on those parts they are really interested in without having to care about other parts of the code.

References

[1] J. E. Jones, The Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature, Proceedings of the Royal Society of London. Series A 106 (1924) 441–462.

[2] L. Verlet, Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules, Physical Review 159 (1967) 98–103.

[3] R. W. Hockney, S. P. Goel, J. W. Eastwood, Quit High-Resolution Computer Models of a Plasma, Journal of Computational Physics 14 (1974) 148–158.

[4] K. Esselink, B. Smit, P. A. J. Hilbers, Efficient Parallel Implementation of Molecular Dynamics on a Toroidal Network. Part I. Parallelizing Strategy, Journal of Computational Physics 106 (1993) 101–107.

[5] M. Horsch, J. Vrabec, M. Bernreuther, S. Grottel, G. Reina, A. Wix, K. Schaber, H. Hasse, Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics, The Journal of Chemical Physics 128 (2008) 164510.

[6] M. Bernreuther, M. Buchholz, H.-J. Bungartz, Aspects of a Parallel Molecular Dynamics Software for Nano-Fluidics, in: G. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bücker, P. Gibbon, B. Mohr (Eds.), Parallel Computing: Architectures, Algorithms and Applications, volume 38, International Conference ParCo 2007, pp. 53–60.

[7] M. Bernreuther, H.-J. Bungartz, Molecular Simulation of Fluid Flow on a Cluster of Workstations, in: F. Hülsemann, M. Kowarschik, U. Rüde (Eds.), Proceedings of the 18th Symposium Simulationstechnique (ASIM 2005), volume 15 of *Fortschritte in der Simulationstechnik - Frontiers in Simulation*, SCS European Publishing House, 2005, pp. 117–123.

[8] M. Buchholz, Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen, Ph.D. thesis, Technische Universität München, 2010. To be published.

[9] G. Karypis, S. K., V. Kumar, PARMETIS - Parallel Graph Partitioning and Sparse Matrix Ordering Library, University of Minnesota, Department of Computer Science and Engineering, 2003.