International Journal of Modern Physics C, Vol. 8, No. 5 (1997) 1131–1140

IMD: A SOFTWARE PACKAGE FOR MOLECULAR DYNAMICS STUDIES ON PARALLEL COMPUTERS

J. STADLER*, R. MIKULLA and H.-R. TREBIN
Institut für Theoretische und Angewandte Physik (ITAP)
Universität Stuttgart, Pfaffenwaldring 57, 70550 Stuttgart, Germany

Received 20 June 1997
Revised 24 June 1997

We report on implementation and performance of the program IMD, designed for short range molecular dynamics simulations on massively parallel computers. After a short explanation of the cell-based algorithm, its extension to parallel computers as well as two variants of the communication scheme are discussed. We provide performance numbers for simulations of different sizes and compare them with values found in the literature. Finally we describe two applications, namely a very large scale simulation with more than 1.23×10^9 atoms, to our knowledge the largest published MD simulation up to this day and a simulation of a crack propagating in a two-dimensional quasicrystal.

Keywords: Molecular Dynamics; Short Range Forces; Massively Parallel Computers; Large Scale Simulations; Cell Algorithm.

1. Introduction

Molecular dynamics (MD for short) simulations are a powerful and widely used tool in statistical physics, chemistry and material science.[1] In MD, a classical many-particle system together with a suitable particle interaction are modeling a fluid or a solid. While conceptually simple, these simulations are computationally extremely demanding. On one hand, a large number of particles is required, on the other also a long simulation time (i.e., many time steps). Both factors add to the appetite for computer resources.

Not surprisingly, much work has been devoted to the development of fast and efficient MD algorithms. Moreover, these were adapted and augmented with each new generation of computers, in order to make the best use of the contemporary machines. For a recent review see Ref. 3.

In this article, we describe the molecular dynamics software IMD,† which is designed for massively parallel computers like the Cray T3E or the Intel Paragon. IMD implements a cell-based, scalable algorithm for short-range molecular dynamics. It employs the SPMD programming paradigm, is written in C and uses the

* *Correspondence to:* Jörg Stadler, Institut für Theoretische und Angewandte Physik, Pfaffenwaldring 57, D-70550 Stuttgart, Germany. E-mail: joerg.stadler@itap.physik.uni-stuttgart.de.
† IMD stands for ITAP Molecular Dynamics.

MPI message passing library. We discuss the implementation of IMD and report performance numbers.

2. Computational Task

In MD the task is to solve the classical equations of motion for a system of N particles over a time τ. The integration over time is done stepwise with a small time step Δt. At each time step the force F_i acting on particle i must be calculated. The interaction between two particles is modeled by a pair potential $\Phi(r)$. It turns out that the calculation of the forces is the most time consuming part of the simulation. We restrict ourselves to *short-range* force laws, where two particles interact only when they are closer than a cut-off distance r_c. In that case, each particle interacts on average with $N_{\text{neigh}}^{\text{sphere}} = \frac{4\pi}{3}\rho r_c^3$ others (ρ being the density).

In the short range case considered here the interaction between particles is local and the problem lends itself naturally to parallelization by domain decomposition. Using domain decomposition has a long tradition in MD simulations. It has been applied on monoprocessor systems by several authors (for a recent list, see Ref. 3) to find pairs of interacting atoms. In MD, these algorithms are known as cell-based schemes.

2.1. *Basic algorithm*

The total force F_i acting on particle i is the sum over all forces f_{ij} that other particles exert on particle i. To calculate F_i we must know the positions r_j of all particles that are closer than r_c to particle i. To find them, we use a geometric algorithm based on the linked-cell method: A parallelepiped with periodic boundary conditions forms the simulation box. It is subdivided into cells (see Fig. 1 for a two-dimensional example). The atoms are assigned to the cells according to their positions. The cell dimensions (i.e., the distances of opposite cell faces) are chosen

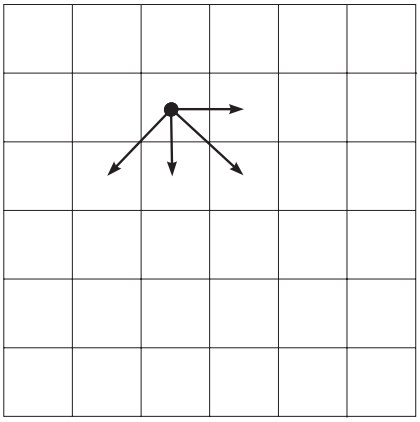

Fig. 1. Decomposition of the simulation box into cells.

such that only particles of adjacent cells can be closer than r_c and an integer number of cells is covering the simulation box in each direction. To calculate F_i we need to check the particles in its own cell and in the 13 surrounding cells (as indicated by the arrows in Fig. 1 for the two-dimensional case). Since $f_{ij} = -f_{ji}$ by Newton's third law, we must only consider each pair of atoms once and thus only half of the neighbors of a given cell. If, for simplicity, we take the case of cubic cells with dimension r_c, for each particle we calculate it interaction with, $N_{\text{neigh}}^{\text{cell}} = \frac{27}{2}\rho r_c^3$ others on average.

IMD allocates memory for each cell such that the coordinates of the cell's particles are stored in a contiguous block of memory. This organization has two main advantages: First, it allows IMD to access memory in a linear, stride-one fashion during the force calculation, and second, it allows us to send particle data of one cell to other processors in a single MPI operation. The main drawback is that this scheme results in a force calculation with a very short innermost loop. This loop, which dominates the overall run-time, has iteration counts of about 30 in the benchmark case. IMD's performance on vector-parallel machines, like the NEC SX4, is therefore poor.

2.2. *Extension to parallel computers*

The extension of our cell-based scheme to parallel computers is straightforward: IMD assumes that the processors are arranged on a three-dimensional cartesian grid (with periodic boundaries), so each processor has 26 neighbors. We use MPI to establish this topology. Our algorithm subdivides the array of cells in equally sized parts which then are assigned to the processors. Since we assume that the underlying physical system is homogenous, this amounts to assigning each processor roughly the same number of particles. Figure 2 shows a two-dimensional example of how the cells are associated with processors: A 12×12 array of cells (shaded in the figure) is distributed within a 3×3 grid of processors. A layer of empty cells (non-shaded in the figure) is added to each processor. These are used as temporary storage for particle data received from the neighbors. We call these extra cells *buffer* cells. The outermost layer of non-empty cells, which forms the surface of a processor's part of the cell array, is called layer of *surface* cells. Particles in non-surface cells interact only with particles on the same processor, whereas those in surface cells also interact with particles in surface cells of the neighboring processors. To calculate the forces for them, data from the surface cells are copied to the buffer cells of the neighbors, as indicated by the arrows in Fig. 2. Let us denote with T_{comm} the time needed to move particle data between processors and with T_{calc} the time used for force calculation and time integration. In other words, T_{calc} measures the useful work done, while T_{comm} measures the overhead due to communication. T_{calc} is proportional to the total number of cells, whereas T_{comm} is proportional to the number of surface cells. We can vary the ratio $T_{\text{comm}}/T_{\text{calc}}$ by changing the number of cells on a processor. When we assign more cells to a processor, T_{calc} will

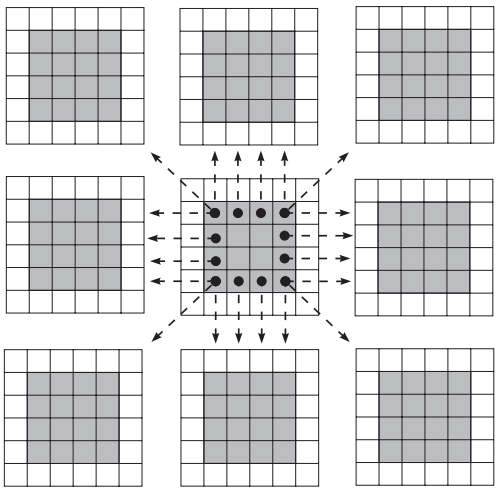

Fig. 2. Assignment of cells to processors, a two-dimensional example. The white cells are the *buffer cells*, the adjacent layer of cells (dotted in the center processor) are the *surface* cells.

increase faster than T_{comm}, since the relative amount of surface cells decreases. The number of cells on a processor corresponds to the number of local particles N/N_p, so $T_{\mathrm{comm}}/T_{\mathrm{calc}}$ depends on N/N_p.

The key points to consider for a parallel MD algorithm are communication, scalability and load-balancing. As we pointed out, we can adjust the relative communication overhead by changing N/Np. By the same argument we see that the run-time should not change when we scale N and N_p such that N/N_p remains constant. We assign roughly the same number of particles to each processor and therefore expect that there is only little loss due to synchronization and that load balancing is good *a priori*. But it is still desirable to execute only a small number of communication operations per time step, since then the processors can run for a longer time without synchronizing, thereby averaging over minor work inequalities.

2.3. *Communication*

Three communication steps have to be performed at each time step: First, the particle positions have to be exchanged with the neighboring processors for the force calculation, second, the forces accumulated due to Newton's third law by particles in the buffer cells are sent back to the original processor, and third, after the positions have been updated, all atoms that have left or entered a processor's volume must be transferred to their new processor. IMD uses the standard MPI library[2] for interprocessor communication. As indicated by the arrows in Fig. 2, data from each surface cell have to be sent to at least one neighbor at each time step. If we communicate data of each cell with one MPI call, we create a large number of small messages. Instead, one can collect all the data bound to a given neighbor first in an intermediate buffer and then transfer the buffer with only one

single MPI call. We have used both approaches — direct and with intermediate buffers — and will discuss them in more detail.

2.3.1. *Direct cell communication*

As indicated by the arrows in Fig. 2, the simplest communication scheme is to directly send the particle position data from the surface cells to the buffer cells of the neighboring processors with one call to the MPI library per cell. We denote this communication scheme as *direct scheme*.

Consider the case where a cubic array of cells with dimension c is assigned to each processor. In three dimensions this algorithm will then perform $6c^2 + 12c + 8$ MPI calls (one for each cell face on the surface, one for each edge and one for each corner). Typical values of c lie in the range from 5 to 50, so from some hundred up to several thousand communication calls are generated at each time step. Moreover, IMD's cell-to-cell communication is implemented in a *save* (see Ref. 2 p. 82ff) way, i.e., it does not rely on buffering supplied by the message passing library. For example, the size of the receiving buffer cell is checked and, if necessary, enlarged to make sure that it can hold the arriving data before the actual send operation is performed. This, as well as the large number of communication calls add to the overhead but also to the robustness of the direct scheme.

2.3.2. *Buffered scheme*

The introduction of communication buffers improves the situation appreciably. Atoms with a common destination are first copied from their cells to a buffer and then sent in just one operation. Plimpton has shown in Ref. 6 that only *six* communication calls per time step (one for each face of a cube) are needed. This number is independent of the number of cells on a processor. But Plimpton's scheme requires some overhead for the maintenance of the buffers, so we choose a simpler approach. We use a buffer for each neighbor and collect all data with a common destination. In that case we have to perform 26 communication operations per time step. We also have to setup and maintain the send/receive buffers and we have to transfer the atom data from the surface cells to the send buffers and from the receive buffers to the buffer cells. The process of moving data between the cells and send- or recieve-buffers is overlapped with the actual communication whenever possible. The drawbacks of this scheme are firstly, that it uses extra memory for the communication buffers, which is, as we will show, an important issue for very large scale simulations, and secondly, that the size of the buffers is fixed for most of the time and only adjusted periodically. Buffer overflows, that terminate the simulation, are therefore possible. Fixed size buffers could have been avoided with different implementation, but we felt that checking the buffer size at each time step would be too much overhead, as it would introduce additional communication.

Tables 1 and 2 show the percentage of time used for communication with respect to the total run-time for buffered and direct scheme. The numbers clearly show the superiority of the buffered scheme.

Table 1. Relative amount of communication with buffered scheme.

N	N_p	N/N_p	Machine	% Comm.
10,976	8	1,372	Cray T3E	9.3
10,976	16	686	Cray T3E	14.7
10,976	32	343	Cray T3E	21.8
10,976	64	172	Cray T3E	33.5

Table 2. Relative amount of communication with direct scheme.

N	N_p	N/N_p	Machine	% Comm.
10,976	8	1,372	Cray T3E	20.4
10,976	16	686	Cray T3E	27.0
10,976	32	343	Cray T3E	35.1
10,976	64	172	Cray T3E	45.4

2.4. *actio=reactio*

When the interaction between a pair of cells, say a and b is calculated, atoms in both cells accumulate forces due to *actio=reactio*. If, however, the cells reside on different processors, say a on processor A and b on B, the atom positions are copied from a to the buffer cell $\hat{a}$ on B and from b to $\hat{b}$ on A respectively. On processor A not the actual atoms in b accumulate the forces, but only their copies in $\hat{b}$. We have two choices in this situation: First, we can calculate the forces for the pair $\hat{a}b$ on B *and* for $a\hat{b}$ on A, or, second, we can calculate only $\hat{a}b$ on B and then send the forces accumulated in $\hat{a}$ to A, where they are added to the atoms in a. The first choice means additional and redundant calculations since we do not exploit *actio=reactio* across processors in that case. The second choice introduces additional communication. We choose to use *actio=reactio* wherever possible, even over processor boundaries. Whether this approach or the redundant calculation is better, depends on the relative speed of communication and calculation on a given machine and also on simulation parameters like r_c or the complexity of the force law employed. Our measurements for a monoatomic Lennard-Jones system show, that using *actio=reactio* is always worthwhile as long as the fast, buffered communication is used. With the slower direct cell communication scheme however, there are some cases where it is better to avoid the additional communication. Since the exploitation of Newton's 3rd law across processor boundaries is favorable in some cases but unfavorable in others, it is a user-selectable option in IMD.

3. Large Scale Simulations

Consider a cube of atoms, say with 1,000 atoms per edge. This is a tiny cube in the real world, its edges measure about one micrometer, but it is a large cube

for an MD simulation containing 10^9 atoms. So the question is, how large can MD simulations be? Can we tackle problems of mesoscopic or even macroscopic scale? How many atoms fit in a given machine's memory? When we consider single precision arithmetic each number uses four bytes of memory. For each atom, we have to store at least position, momentum and force. So, in three dimensions, each atom uses 36 bytes of memory. To simulate the cube discussed above, one needs 36 GB. The T3E installed at HWW in Stuttgart has 512 nodes with 128 MB per node for a total of 64 GB. Of course, only a part of it can be used to store atom coordinates. Considerable space is used by code and data of operating system and also by code and internal data structures, like e.g., buffer cells, of IMD. We were able to simulate 1,213,857,792 atoms on this machine, thereby using roughly 44 GB. The direct cell communication scheme was used to save memory. It took 30 minutes to complete 10 time steps. Details of this simulation are summarized in Table 3. To our knowledge, this is the largest MD simulation reported up to today.

Table 3. Performance numbers.

Groups	Machine	N	N_p	t_{step} s	$t_{\text{pair}}^{\text{one}}$ μs
current work	T3E	296,352	8	1.956	0.588
current work	T3E	1,000,188	27	1.960	0.589
current work	T3E	2,370,816	64	1.962	0.590
current work	Paragon	2,370,816	64	22.552	6.780
current work	SR2201	296,352	8	3.675	1.105
current work	T3E	1,213,857,792	512	180	1.61
Lohmdahl *et al.*	T3D	75,000,000	128	46.9	1.46
Plimpton	Paragon	100,000,000	3680	3.5	2.34
Lohmdahl *et al.*	CM-5	600,000,000	1024	242	7.51

4. High Speed Simulations

MD simulations can be large in two ways: The first, discussed above, is that one often needs a large number of particles. This case is quite easy to parallelize since it puts a large number of atoms on each processor, and, as mentioned earlier, the relative communication overhead goes down as N/N_p increases. The second is that MD simulation often also requires a large number of time steps and therefore should be as fast as possible. If we want to increase the speed of a simulation of given size, we would like to use as many processors as reasonable. Adding processors means decreasing N/N_P while N remains constant, so we expect the overhead due to communication to increase. Table 3 shows the time used for communication for different N/N_p. On the T3E, IMD needs about 600 atoms per processor to keep the relative amount of communication below 30% of the total time. In that example,

a system of 296,352 particles was propagated for 15 time steps per second on 512 processors.

5. Timing Results

Now that we have presented the general features of IMD, we want to discuss performance in more detail. The speed of an MD simulation depends on the force law employed. It is common practice in literature[3,6] to use a monoatomic Lennard-Jones system as a benchmark problem to compare algorithms. Several measures can be given to characterize the performance: Various numbers are used in literature to characterize performance: t_{step} and $t_{\mathrm{particle}} = t_{\mathrm{step}}/N$ are the time used per time step and the time used per time step and particle, respectively. t_{step} gives a direct feeling of how many time steps can be done in a second, but depends on the system size, t_{particle} is independent of system size, but still depends, as t_{step}, on r_c and on N_p. Beazley *et al.*[3] suggest the *effective time per pair interaction* should be defined as $t_{\mathrm{pair}} = t_{\mathrm{step}}/(\frac{4}{3}\pi r_c^3 \rho N) = t_{\mathrm{particle}}/\frac{4}{3}\pi r_c^3 \rho$ and the *one-processor effective time per pair interaction* defined as $t_{\mathrm{pair}}^{\mathrm{one}} = N_p t_{\mathrm{pair}}$ for performance comparisons. Both numbers are independent of r_c; $t_{\mathrm{pair}}^{\mathrm{one}}$ is also independent of N_p. They represent the time used to calculate, in double precision, single Lennard-Jones interaction between a pair of atoms and thus allow to compare the speed of simulation runs with different r_c and even different N_p. Table 3 compares IMD for simulations of different sizes and on different machines with some of the more recent performance data collected in Ref. 7.

We will now discuss the scaling behavior of IMD. Our results are shown in Tables 4 and 5. Here we distinguish two cases: First we simultaneously increase the system size and the number of processors, such that $N/N_P = const.$ According to the scaling argument given above, the run-time should be constant. Our measurements are presented in Table 4. We find, as we expect, nearly perfect scaling for 3^3 and 4^3, but a notable decrease of efficiency for 6^3 and 8^3 processors. In the second case we increase N_P while we keep N constant. Table 5 shows how the overhead due to communication grows with decreasing N/N_P.

Table 4. Scaling with $N/N_P = const.$

Machine	N	N_P	N/N_p s	t_{step} μs	$t_{\mathrm{pair}}^{\mathrm{one}}$	% Efficiency
T3E	296,352	8	37,044	1.956	0.588	100.0
T3E	1,000,188	27	37,044	1.960	0.589	99.7
T3E	2,370,816	64	37,044	1.962	0.590	99.6
T3E	8,001,504	216	37,044	2.161	0.650	90.5
T3E	18,966,528	512	37,044	2.479	0.745	78.9

Table 5. Scaling with $N = const$.

Machine	N	N_P	N/N_p	t_{step} s	$t_{\text{pair}}^{\text{one}}$ μs	% Efficiency
T3E	296,352	8	37,044	1.956	0.588	100.0
T3E	296,352	27	10,976	0.593	0.600	97.7
T3E	296,352	64	4,630	0.258	0.621	94.7
T3E	296,352	216	1,372	0.101	0.820	71.7
T3E	296,352	512	580	0.063	1.201	48.5

6. Example

As a general-purpose MD program, IMD was not developed with a special application in mind. Currently it is used for a variety of problems. An example is the study of crack propagation for two-dimensional models of quasicrystalline materials.[7] Figure 3 is a snapshot of a simulation run on 64 processors. The rectangular strip shown consists of roughly 250,000 atoms of a two-dimensional model quasicrystal. The atoms are too dense to be depicted individually in this case, so we choose to color-code the kinetic energy in the picture. Low values are represented by dark colors, high values by light colors. The system starts with an initial temperature close to zero. At the crack tip, bonds break during the crack's movement thereby they release energy into the system and generate sound waves, which are clearly visible in the figure. To avoid reflection of the waves from the boundaries an elliptical stadium is created outside in which the waves are damped gradually by a ramped friction term in the equations of motion.[8] On the T3E, such simulations are finished in just a few hours, which allows us to study a wide range of situations and parameters.

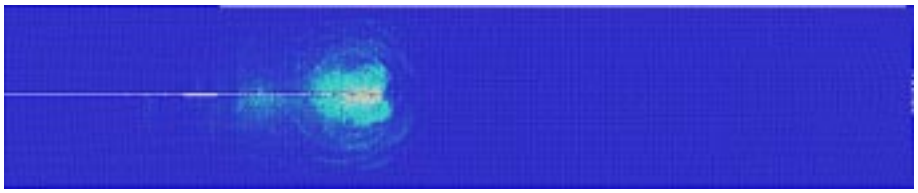

Fig. 3. Snapshot of crack simulation.

7. Conclusion

Massively parallel computers add another dimension to the design of numerical programs: The speeds of the memory and the processor are no longer the only factors to consider. Also the speed of interprocessor communication must be taken

into account. With the message passing library *MPI*, that allows us to control the communication explicitly, we developed our code IMD for short-ranged molecular dynamics simulations, that tries to find reasonable balance between the three factors. One version of the code, that implements the direct cell communication scheme, is suitable for very large scale simulations, while the other, using buffered communication, speeds up smaller calculations considerably. Machines like the Cray T3E are mighty tools for MD: With the large distributed memory (64 GB), we could demonstrate a simulation with roughly 1.2 billion atoms using only three minutes per time step. To our knowledge, this is the largest simulation reported up to now. But while its size may be impressive, it is the speed that makes the parallel machines so valuable for research: Our simulations on crack propagation used to take several days on fast workstations. Now, running on 64 processors of the T3E, they are finished in just thirty minutes.

Acknowledgments

This work has been funded as *Teilprojekt A2* of the *Sonderforschungsbereich 382* by the Deutsche Forschungsgemeinschaft. This support is gratefully acknowledged.

The authors wish to thank J. Roth, P. Gumbsch, R. Niemeier, and J. Bachteler for inspiring and helpful discussions.

References

1. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford Science Publications, 1987).
2. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI* (The MIT Press, 1994).
3. D. M. Beazley, P. S. Lomdahl, N. Gronbech-Jensen, R. Giles, and P. Tamayo, *Ann. Rev. Comput. Phys.* **3** (World Scientific, 1995).
4. D. M. Beazley and P. S. Lomdahl, *Parallel Comput.* **20**, 173–195 (1994).
5. K. Esselink, B. Smit, and P. A. J. Hilbers, *J. Comput. Phys.* **106**, 101–107 (1993).
6. S. Plimpton, *J. Comput. Phys.* **117**, 1–19 (1995).
7. R. Mikulla, J. Stadler, P. Gumbsch, and H.-R. Trebin, to appear in *Proc. of the 7th Int. Conf. Quasicrystals*, Tokyo (1997).
8. S. J. Zhou, P. S. Lohmdahl, R. Thomson, and B. L. Holian, *Phys. Rev. Lett.* **76**, 13, 2318 (1996).