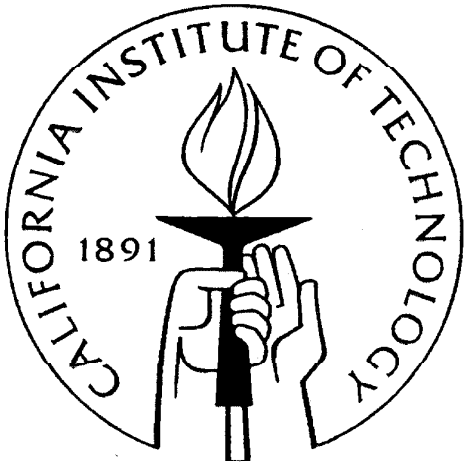

Molecular Dynamics on the Mosaic

Klaas Esselink
Jan L.A. van de Snepscheut

Computer Science Department
California Institute of Technology

Caltech-CS-TR-92-25

Molecular Dynamics on the Mosaic

Klaas Esselink
Shell Research
Amsterdam

Jan L.A. van de Snepscheut
Computer Science, Caltech
Pasadena

Abstract

We describe the development, implementation and performance of a parallel Molecular Dynamics program for the MOSAIC network of processors being developed at Caltech. A communication protocol is set up, and the influence of using different communication networks on the physical network is studied.

1 Introduction

Molecular Dynamics simulations (MD) have become a popular means to study the behavior and properties of systems of particles. The simulations are CPU intensive, and there is a need to simulate ever larger and more time consuming systems. Much of the computation however can be parallelized. The power of parallel computers makes possible large computer experiments, much larger than what is feasible with a state of the art single processor computer, even today. Parallel computers will keep this advantage as long as the processors they consist of can communicate with each other efficiently.

It is relatively easy to use coarse grain parallel computers, systems in which there are not many, yet very powerful, computing nodes. It is the intention of this report to show that also with fine grain computers such as the Mosaic system (being developed at Caltech) useful simulations, like Molecular Dynamics, can be done. Moreover, the implementation can teach us something about properties

of networks with the same structure as the Mosaic, now that some of the more recent commercially available systems are built using similar routing concepts.

After an introduction to Molecular Dynamics in section 2, aspects of a parallel implementation are discussed in section 3. Section 4 briefly describes the Mosaic, as well as some difficulties in handling the communication routines. This results in the development of a new communication protocol. Section 5 compares the performance of the implementation to implementations on other platforms, and it studies the effect of defining different topologies on the physical mesh. Section 6 contains some details of the implementation, and section 7 ends with some conclusions.

The work reported here was carried out in March and April 1992 at the Computer Science department of Caltech. Some experiments were done on the $4 * 16$ processor Mosaic network available then, while other and larger experiments were completed in November 1992 on a network of $16 * 16$ processors. The implementation on the Mosaic was ported from an earlier version developed at Shell Research (Amsterdam) on Transputers by Peter Hilbers and Klaas Esselink [2]. We thank Peter Hofstee for carefully reading this manuscript.

Throughout this report, the term 'Transputer' refers to the T800 version.

2 Molecular Dynamics

The aim of Molecular Dynamics (MD) is to study the macroscopic behavior of systems of particles by simulation at a semi-atomic scale. Newton's equations of motion are integrated over time for all particles. The forces on the particles are determined by the interaction potentials. From the position and velocity of and forces on all particles, many properties of the system, like temperature, pressure, distribution functions etc, can be calculated. In general, the systems are made periodic, to reduce size and boundary effects.

The main 'parameters' of the simulation are the potentials used to model atomic interaction. With respect to the electromagnetic force two particles exert on each other as a result of their charge, the well-known Coulomb potential can be used. This potential is 'long range', meaning its influence stretches over a reasonably long distance (more than half the box length). The Van der Waals atom-atom interaction is usually modeled by the Lennard-Jones potential (deduced from the dipole-induced dipole interaction):

$$LJ_{ij} = 4\epsilon((\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^{6}). \tag{1}$$

This potential has two parameters, σ, the collision diameter (or size of the atoms), and ϵ, the depth of the potential well at the minimum; r_{ij} is the distance from particle i to particle j. The potential provides a reasonable description of the interaction between pairs of atoms in rare-atom gases, and of pseudo-atoms like CH_4. Many experiments have been done to determine the properties of

Lennard-Jones fluids [4]. They are well suited for a qualitative and sometimes for a quantitative comparison with real systems.

Although the Lennard-Jones potential is merely an approximation, it is used extensively. The main reasons are that it has been applied successfully to some systems, and that it is easy to implement. It is no secret that the twelfth power is chosen because it is the square of the sixth. For many applications, it can be safely cut-off at for instance $R_c = 2.5\sigma$.

Other important potentials are those used to model molecules. We have the bonding potential (to couple two particles):

$$BO_{ij} = \frac{1}{2} B_c (r_{ij} - B_l)^2, \tag{2}$$

the bending potential (to describe the angle potential between two bonds):

$$BE_{ijk} = F_{\mathrm{BE}}(\theta_{ijk}), \tag{3}$$

in which θ_{ijk} is the angle between $\vec{r}_{ji}$ and $\vec{r}_{jk}$ in the chain (i, j, k), and the torsion (to describe the dihedral angle potential):

$$TO_{ijkl} = F_{\mathrm{TO}}(\tau_{ijkl}), \tag{4}$$

in which τ_{ijkl} is the angle between the planes ijk and jkl.

The Lennard-Jones, bonding, bending and torsion potentials are short-range. Assuming constant density and a homogeneous distribution of particles, the amount of work done for each particle does not depend on the total number of particles in the universe. Implementations of order $O(N)$ are therefore feasible, exploiting the 'locality' of the computations.

A complete Molecular Dynamics program repeatedly calculates the forces on all particles, and from those the accelerations, velocities and positions. There are several ways to integrate time, this report does not describe the various methods, please see [1] for a thorough description. Let it suffice to say that we use the Leap-Frog integration. Furthermore, we will not deal with any long range force, all potentials to be used must be zero beyond a certain cut-off R_c.

A general property of the particles in a simulation is that they do not move very fast, which is a direct result of the fact that the time step should be sufficiently small in order to guarantee accurate time integration. As a result, it pays to keep track of all particles in a particle's neighborhood. This list can then be used for several iterations before it needs to be updated. Verlet [10] was the first to report this implementation, and these Verlet lists are frequently applied since.

For this project, the information mentioned above suffices. More details about the Molecular Dynamics technique can be found in, e.g., [1].

3 Parallel Molecular Dynamics

There are a few techniques to parallelize computations. In [3] we argue that geometric parallelism can be advantageous, given the fact that the evaluation of the Lennard-Jones constitutes the bulk of the work. In geometric parallelism, a processor performs the relevant computations confined to a particular area of the simulation universe. In the case of Molecular Dynamics, this means that only the movement of the particles in a processor's space needs to be calculated by the processor. It also means that communication of relevant particle information can be restricted to a known set of, preferably close-by, processors. The cost of communication can thus be kept low compared to the computation cost. In fact, one can prove that, if the processor network is a torus and not 'too' large [3], the best way to minimize communication is assigning equally large columns of the simulation universe to the individual processors, and neighboring columns to neighboring processors. Please note that the need for mapping neighboring columns to neighboring processors arises from the fact that communication cost is assumed to increase with distance in the processor network. This assumption is valid for a Transputer network without routers, but it is not valid for Mosaic networks, and we will use this by using communication networks different from the physical networks (subsection 5.1).

3.1 The Program

Using geometric parallelism on a toroidal network, the program of figure 1 can be executed on each processor.

```
initialize;
GOON:=continue;

while not_finished and GOON do
 begin integrity;
        if continue
          then move_it
          else GOON:=false
 end;

finalize;
```

Figure 1: The main program

Procedure `initialize` reads and distributes the parameters of the simulation, as well as particle information. Each processor determines which part

of the universe is his, and receives the corresponding particles. This is accomplished by having all particles sent over a ring, a processor takes off whichever particles are his.
Function **continue** checks the error status of all the processors. Since it is not possible for the individual processors to inform and stop other processors whenever an error occurs, errors have to be logged until they are handled by **continue**.
Procedure **integrity** checks the positions of all particles in a processor's space. During the simulation, particles can cross processor boundaries, and they will change processors. Upon termination of **integrity**, each particle resides in the right processor.
Procedure **move_it** calls a particle communication procedure **compos_id** and evaluates the forces. Using Newton's third law, all forces are evaluated only once, which means that they have to be communicated back also. After this, all accelerations, new velocities and new positions are computed.
Procedure **compos_id** acquires copies of the particles of 4 neighboring processors. With the following communication scheme, there is no danger of deadlock (see figure 2). First, information is sent to the east (and simultaneously received from the west, flow 1), then to the south (flow 2). Next, information from the north-west is needed, but this has just been sent to the east, so it can be found at the north processor (flow 3). Likewise, information from the south-west can be found at the south processor (flow 4). Note that it is important to perform **integrity** before **move_it**. Note also that the size of the columns on a processor should be at least the cut-off R_c in the x- and y-direction, otherwise information will be missed. If R_c exceeds the size of the columns, a different communication scheme is required. The present Mosaic implementation requires R_c to be at most the column size.

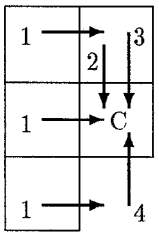

Figure 2: A processor ('C') and the flow of particle information

Forces are communicated by **comfor**, exactly the other way around. Note that for the communication of the positions the data of a particular particle needs to be replicated, while for the forces it needs to be accumulated.

More specific information about the implementation can be found in section 6.

4 Implementation on the Mosaic

4.1 General

A Mosaic chip integrates a processor, 64 Kbytes of private memory and a router. The chips can be used to build a mesh network in which the routers offer a point-to-point connection between any two processors. Effort has been put into making this communication very fast, in fact, communication from processor to processor is faster than an intra processor block move. Further details about the chip and the project can be found in e.g. [8] and references therein.

One Mosaic chip is the basic building unit for a preferably rectangular mesh of processors. Using the fast point-to-point connection, we first use a toroidally connected communications network mapped straight on the physical processor network, in order to implement the periodic boundary conditions. The columns of the universe will be mapped such that neighboring columns are on neighboring processors. (Whether or not this avoids communication congestion is tested in section 5).

There is one basic rule while using inter processor communication: *always* be prepared to read an incoming message. Failure to do so may result in not being able to send a message and is a main cause of deadlock. This rule plays an important role in all implementations. We will (have to) pay special attention to it.

4.2 Programming language

The language used in this project is Mosaic Pascal [5]. It consists of standard Pascal with a few extensions to support parallelism and also a few omissions. One of the most important new concepts is that of multiple processes per processor. These processes can communicate with each other via channels (using the commands 'read' and 'write'). The first one is cobegin/coend:

```
cobegin statement_0;
        statement_1;
        ...
coend
```

in which the mentioned statements are executed in parallel; a typical use would be to send and receive data at the same time. (See procedures `integrity`, `compos_id` and `comfor`, subsection 3.1.) The second way is fork/join:

```
fork proc_0: aclass;
proc_1;
```

```
...;
join aclass
```

in which procedure `proc_0` runs as an independent thread simultaneously with the forking thread. The join statement waits for the completion of all threads forked in a certain 'class'. Monitoring processes or buffers would typically run in parallel with the main program. (See procedure `Custodian`, introduced in subsection 4.4.)

As mentioned before, intra processor communication is done over channels. These channels have zero slack, the two processes reading from and writing to a channel are synchronized. Since inter processor communication may not necessarily have the same properties, no attempt was made to offer channels to communicate with other processors. Instead, the commands 'send' and 'receive' are provided. In their most bare form, 'send' would take a destination processor and a message address as a parameter, while 'receive' needs only a destination address. This is in fact the model supported by the (routing) hardware. In practice however, the programmer would then end up having to write some sort of multiplex/demultiplex if she ever would like to use multiple processes per processor. For this reason, the notion of links is introduced. The programmer may use any number of links per program, and messages enter a processor over a specified link. This link number is therefore also a parameter to 'send' and 'receive'. Dedicating a certain link to a particular process would be good programming practice, in order to avoid having two processes reading from the same link simultaneously.

4.3 Controlling Send and Receive

At the lowest (i.e. machine) level, the arrival of a message is signaled by an interrupt. Each message is augmented with a header by Pascal's 'send' to indicate which link it should appear on, and during the arrival an interrupt occurs as soon as the header is read. The Pascal interrupt handler next checks to see what link the rest of the message should appear on, it takes the address of the destination variable for that link and acknowledges the interrupt. This signals the hardware that the rest of the message can come in. If the interrupt is not acknowledged, the rest of the message will not be read and stay pending in the network. In this situation, any 'send' from that processor is not guaranteed to succeed. This means that the following piece of program:

```
send(DestinationProcessor, DestinationLink, data);
receive(SourceLink, variable)
```

is not guaranteed to terminate if the message to be read is already waiting to come in. This is a direct result of the fact that if the message arrives and the corresponding receive has not been executed yet, the interrupt handler doesn't

know where to place the message. Buffering on a processor with so little memory is out of the question, so the interrupt will not be acknowledged, and the preceding send may not terminate. The problem gets even more complicated when more than one process is active, since the scheduling of processes can only be done on the basis of an interrupt or communication over channels; there is no time slicing. If a receive interrupt is not acknowledged, no new ones will be generated, which means that scheduling the wrong process may result in deadlock.

On machines with enough memory, these problems are avoided. Messages are read in as soon as they arrive by allocating a new buffer, the running program gets a pointer to the message at the moment it asks for it (compare e.g. the reactive kernel [9]). This approach may run into some memory problems if the receiving of messages is not controlled one way or the other and the available memory is limited. A solution would be to issue a send command only after being sure that the receiving processor is ready to accept it. This means that some sort of (software) protocol is needed, in which the sending of a message is preceded by a request to do so. Furthermore, we have seen that upon arrival of a message, the corresponding receive must be pending. One way to do so is to have as many request buffers as there are off-processor sending processes. It is then possible to have the main program acknowledge such a request at the moment it is ready to actually receive a message from such a sending process. This approach limits the number of processors that can be communicated with, as a direct result of the limited memory. The power of the routing hardware, which actually permits communication between any two processors, needs to be limited to communication with a known maximum number of processors in order to be able to write programs that are guaranteed deadlock free.

For the record we would like to remark that these problems are unknown to networks of Transputers if one uses only neighbor communication. It is perfectly valid then to leave an incoming message unread at one link, while reading or sending a message over an other. The 4 links are physically independent.

4.4 Communication Protocol

In order to use 'send' and 'receive' correctly, we have shown that we must limit ourselves to communication with a fixed number of off-processor processes. Let us call these different processes directions. Furthermore, we will need some sort of protocol to guarantee that upon arrival of a message, the corresponding receive instruction is pending (already). First we redefine the send and receive commands as follows:

```
RECEIVE(dir, mes) == begin set_receive(dir.inlnk, mes);
                           CSEND(dir.destproc, dir.custlnk, any);
                           wait_receive(dir.inlnk)
                     end
```

```
SEND(dir, mes) == begin write(dir.sig, any);
                        CSEND(dir.destproc, dir.reclnk, mes)
                  end
```

in which 'CSEND' denotes the 'critical' original function. In order to guarantee that upon arrival of a message the corresponding 'receive' is pending, the 'receive' statement has been split into 'set_receive' and 'wait_receive'. 'Set_receive' assigns a variable to a link, meaning that upon arrival of a message on the link, this message should be put in the variable. 'Wait_receive' stays pending until the message has been read. After setting the variable for a link, 'RECEIVE' takes the initiative for a communication by sending a signal to the other side. There it is received on link 'custlnk' by:

```
procedure Custodian(custlnk: integer; var dirsig: channel);
begin set_receive(custlnk, any0);
      while true do
        begin wait_receive(custlnk);
              set_receive(custlnk, any0);
              read(dirsig, any1)
        end
end
```

which in turn signals the corresponding 'SEND' on channel 'dirsig' (the channels 'dir.sig' of 'SEND' and 'dirsig' of `Custodian` are the same). The communication of the actual message completes the protocol. Provided that `Custodian`'s very first 'set_receive' has been performed before the corresponding message comes in, *all* messages sent arrive *after* the corresponding 'set_receive' has been performed[1].

5 Experiments

With the implementation as explained in section 3 based on the protocol as derived in 4, experiments were done with simple Lennard-Jones fluids. Here, we concentrate on the timing results and leave the actual study of, e.g., a phase diagram to interested physicists. First, we need to define a measure of performance in order to come to a comparison with other machines and implementations. The properties of a simulation system listed in table 1 are common to many simulations and influence the timing results.

Per iteration, the total number of pairs C of particles that needs to be considered for force calculations is

$$C = \frac{2}{3}\pi R_c^3 \rho N \tag{5}$$

[1]The text of RECEIVE, SEND and Custodian as presented here was first written down by Peter Hofstee at one of our Monday morning sessions.

Symbol	Description
N	Number of particles
ρ	Density
R_c	Lennard-Jones cut-off

Table 1: Simulation properties relevant for timings

giving a lower bound for the work that has to be done. The performance G_s of different MD implementations can be compared by taking $G_s = C/\tau$, where τ denotes the (average) time needed per iteration. For parallel implementations on networks of P processors, we take the performance $G_p = C/\tau/P$. Please note that these definitions of performance relate mainly to MD performance.

Many details of an implementation influence the actual timing results. A naive algorithm does not take the short range nature of the potential into account, but just generates all pairs of particles. This algorithm will scale quadratically in N, and therefore perform poorly for larger numbers of particles. As mentioned in section 2, other algorithms generate a list of neighboring particles per particle (Verlet list method). This list is used several iterations, decreasing the average time needed per iteration. A third method divides the universe into cubic cells, and determines the forces by searching particles in the same and neighboring cells only (linked list method). Another method is to combine these two algorithms. This is done in the parallel implementation of [3].

Our parallel Mosaic implementation uses the third method, it distributes columns of cells with edge sizes at least R_c over the processors. If the edge size is much larger than R_c, the communication of and search for neighboring particles will take more time than necessary (since also particles at a relatively large distance are involved). This will have a negative influence on the timing.

In table 2 we find timings and performance measures of several runs, including some taken from the literature.

We would like to make the following remarks about this table. There are three subtables, the first one of runs on the Mosaic, the second of runs on Transputer systems and the third on Intel i860-based systems. The runs on the Mosaic were carried out on the networks connected to SUNs 'mercury' and 'mars'. The runs on systems of size 21632 particles show the effect of larger subcells: on smaller processor networks, the performance G_p degrades somewhat. It is strongly influenced by the relation between the potential cut-off R_c and the size of a subcell. The small low-density systems ($\rho = 0.1$) suffer from a bad load balance (only 12 particles *on average*), which becomes better for smaller networks. Obviously the size of the simulated system is too small here. Using a very small cut-off results in other aspects of the parallel algorithm, like communication, becoming more important. These aspects become less important as

System	#Particles	ρ	R_c	Machine Px*Py	τ (s)	G_s	G_p	Floprate
mercury	7168	0.7	3.57	Mosaic 4*14	16.60	288e2	514	787e2
mars	21632	0.7	2.5	Mosaic 16*16	4.53	109e3	427	955e2
mars	21632	0.7	2.85	Mosaic 16*16	4.80	152e3	597	946e2
mars	21632	0.7	2.5	Mosaic 15*15	5.72	866e2	385	967e2
mars	21632	0.7	2.5	Mosaic 14*14	7.20	688e2	351	999e2
mars	21632	0.7	2.5	Mosaic 13*13	9.55	518e2	307	100e3
mars	21632	0.7	3.57	Mosaic 13*13	13.3	108e3	641	982e2
mars	3136	0.1	2.7	Mosaic 16*16	0.22	587e2	229	440e2
mars	3136	0.1	2.7	Mosaic 15*15	0.24	538e2	239	512e2
mars	3136	0.1	2.7	Mosaic 14*14	0.27	478e2	244	588e2
mars	5324	0.7	1.12	Mosaic 16*16	0.17	645e2	251	528e2
mars	21296	0.7	1.12	Mosaic 16*16	0.46	953e2	372	780e2
mars	26620	0.7	1.12	Mosaic 16*16	0.57	961e2	375	787e2
(R_s=0.5)	21632	0.7	2.5	Transputer 20*20	0.52	953e3	2382	215e3
(R_s=0.4)	21632	0.7	2.5	Transputer 16*16	0.67	740e3	2890	221e3
(R_s=0)	21632	0.7	2.5	Transputer 16*16	2.0	248e3	968	204e3
(R_s=0.78)	26620	0.7	1.12	Transputer 20*20	0.16	343e3	857	145e3
(R_s=0.08)	26620	0.7	1.12	Transputer 16*16	0.23	193e3	754	105e3
(R_s=0)	26620	0.7	1.12	Transputer 16*16	0.40	137e3	536	115e3
[2] (R_s=0.3)	10000	2.2	2.5	Transputer 6*6	6.0	120e3	3333	247e3
[2] (R_s=0.3)	10000	2.2	2.5	Transputer 20*20	0.85	847e3	2117	155e3
[2] (R_s=0.3)	39304	0.7	2.5	Transputer 20*20	0.86	105e4	2617	
[7]	2050000	0.7	1.12	iPSC/860 64	2.69	157e4	24530	
[6]	1000000	0.84	2.5	Delta 512	0.55	500e5	97610	

Table 2: Timing results

the system size increases. For all these simulations, the flop rate was estimated by calculating the number of distance checks (20 flops) and Lennard Jones evaluations (26 flops) on the basis of average numbers of particles per subcell. Also note that, in order to make timings at all possible, specific synchronization had to be introduced. Leaving 'timing' out resulted in an average increase of 3% in speed.

The first six simulations on Transputer networks were done on exactly the same input systems as for the Mosaic. The implementation uses Verlet neighbor lists, of particles within distance $R_c + R_s$ of each particle. Setting R_s equal to 0 degrades the performance effectively to that of the standard linked list method, which should make it comparable to the implementation on the Mosaic. However, the Verlet lists *are* generated in that case, so the reported figure for

that case is somewhat pessimistic and can be improved easily.

The Transputer implementation is capable of handling cells with any size (not necessarily related to R_c). If the cells have size $R_c/2$, the information necessary for the calculation of the force on one particle has to come from 62.5 cells of volume $R_c^3/8$, compared to 13.5 cells of volume R_c^3 if the cell size is R_c. This is considerably less, and our Mosaic implementation cannot take advantage of that (yet). When executing comparable algorithms, the Transputer is 1.46 to 2.14 times faster than the Mosaic. The ratio is larger in MD systems with large R_c as they lead to more floating point operations per communication. (The Transputer is a 32 bit processor with a floating point unit, the Mosaic is a 16 bit processor without a floating point unit.) The Transputer implementation allows R_s to be increased so that it can take advantage of Verlet lists, as well as using cells with edge size $R_c/2$. This algorithmic improvement yields a factor of 1.74 to 3 on Transputers. We suspect that the same algorithmic improvement would lead to a comparable speed improvement on the Mosaic.

5.1 Remapping

In our implementation we have followed the rather conventional approach of mapping adjacent columns of the universe to adjacent processors in the physical network, in other words, the communication network and the physical network are identical. The reason behind this approach is that communication cost should be minimized, and indeed, for processor networks built with Transputers without routing hardware, this is necessary. Nevertheless, given a specific algorithm, one would like to have more freedom in *defining* a communication network, regardless of the physical one. Having fast routing hardware is a necessary tool for that. An application can then assume to have logically close neighbors which are, in fact, physically very distant. In theory, one would expect the communication cost to increase (when the communication and physical network differ) for two reasons, the first one being that larger distances take more time, and the second that the chance of congestion increases. On the Mosaic, we measured no difference between short and long communications. The order to which the second reason is important depends on the communication behavior of an application. For our MD implementation, we tested the importance of running on a physical torus by comparing it to runs on a logical torus, mapped randomly on the physical torus. During the initialization of the MD program, a logical torus is read from a file. This torus is constructed by a separate Pascal program ('mapper.p'), which generates logical tori by repeatedly swapping elements of the physical torus. Using the $16 * 16$ network, we found that the execution time of the MD algorithm does not decrease at all if the logical torus implies longer communication paths in the physical torus. This indicates that the choice of the physical Mosaic network (torus, or rather, mesh) does not influence the performance of the algorithm.

6 Implementation Details

The actual parallel program consists of one Mosaic Pascal program for the net processors ('net.p') and one for the intermediate host processor ('host.p'). 'Nmon' [5] is supposed to run on the SUN host. 'Host.p' uses a program called 'run.p' to boot the network with the right code.

The MD programs 'host.p' and 'net.p' have the same structure. Their source consists mainly of '#include' files, of which there is usually a 'hversion.i' and a 'nversion.i'. Only the declaration of types and variables as well as the body of the main program can be found in the main source.

In what follows we describe the functionality of some of the include-files.

- 'options.i': Compile time options. The following options are implemented:
  - VERBOSE: for extra text during booting,
  - DEBUG: for extra text during the total simulation,
  - FORCE_CHECK: to compare the calculated forces with physical sense,
  - SIGMAFIXED1: if all sigmas are one, 2D array indexing can be avoided,
  - ENERGY_TOO: provide potential and kinetic energies for each time step.
  - TIMING: include timing results.
- '[hn]machine.i': Supposedly, all machine-dependent features should be implemented here. The communication protocol is handled, the logical torus, i.e. the north, east, south and west connections, is set up (if available, it is read from the file 'mapping', see below) and the ring (prev and next) is defined. The host version reads the file 'netusage' (if available), containing three integers: the distance from the host to the net, and the x- and y-size of the network.
- '[hn]general.i': General procedures, including global sums etc.
- '[hn]error.i': Handling of errors should be a synchronized action (since every processor must be involved).
- '[hn]tohost.i': Specific information is gathered by the host.
- '[hn]initialize.i': Simulation parameters are read from the file 'MD'. This file basically consists of pairs of lines, the first one of which is comment and the second one containing the actual data. The data has the following meaning:

TEMPRQ	Requested temperature of the system
DT	Time step for Leap Frog integration
EPSI	Epsilon for Temperature calculations

RL	Used to determine the subdivision in the z-dimension
ITmax	Last iteration number
TS	Temperature Scaling frequency (in iterations)
SAVE	System dump frequency
NCOMP	Particles have a type number, this is the number of types
	Per type, we define:
ID	Type number
X	Mol fraction of that type
Eps,Sig,Rc	ϵ, σ and R_c of LJ interaction between two particles of that type
i,j,Eps,Sig,Rc	Per remaining pair of ID's $(i,j), i \neq j$, the LJ is defined
NSPRING	The number of springs in the system (not completely implemented)

The particle information is read from the file 'MDps', which is mixed ascii-binary. The first line contains the size of the universe (RX, RY and RZ), the number of particles and the iteration counter in ascii. After that, the file contains information per particle (ID, particle number, position and speed) in binary format.

- '[hn]integrity.i': Place particles in the right cell.
- '[hn]compos.i': Communicate neighbor particle positions and identities
- '[hn]comfor.i': Communicate computed forces on neighboring particles
- '[hn]continue.i': Synchronized error check
- 'force.i': Minimum image distance and Lennard-Jones force calculations
- '[hn]move.i': One basic iteration
- 'main.i': The main program

During a simulation, the user can press 'return', which will result in a break of the simulation. A menu of things to do is presented (see 'hcontinue.i'), like immediate dumping of the system, and producing graphical output. The main drawing routines can be found in 'graphics.i'. The format of the produced picture is somewhat obscure, to be read by the SUNVIEW-oriented program 'xplot', and X-output is desirable.

The (SUN-) Pascal program 'mkMDps.p' takes an 'MD' file and generates a universe in 'MDps' based on the assumptions about the size of the network it should run on, R_c, and ρ. Particles are generated on an fcc-lattice.

The program 'mapper.p' generates random tori, which can be used to define a logical torus, different from the physical Mosaic mesh-network. Currently, the logical and physical torus should have the same dimensions, but this is not strictly necessary.

7 Conclusion

We have implemented a parallel Molecular Dynamics program on the Mosaic network, using the familiar approach of one program for every node. The program requires little memory (13 Kbyte), leaving enough space for data on every node. Although floating point operations are not supported by hardware and have to be handled by software, 100 Kflop per second per node is obtained, making a 16,000 node machine (the ultimate size of the Mosaic) a fast one.

The routing hardware proves its usability by showing no performance decrease if the logical torus (on which computations take place) differs from the physical torus. This feature makes applications independent of the physical connection scheme of a machine.

The routing hardware offers a point-to-point connection between any two processors, but for the programmer this functionality is hard to exploit. A communication protocol is set up, providing guaranteed communication between a processor and a restricted set of other processors. We expect this functionality to become more and more important, now that new commercially available parallel computers usually are equipped with routing hardware.

One of the philosophies behind the Mosaic design is that it should be possible to view the Mosaic not so much as processors with some memory, but rather as memory with a small (control-) processor. Small pieces of code could be sent to a processor together with the relevant data, or, processors could serve memory requests from other processors only. This approach leads to a new programming concept, in which parallelism is obtained at a different level, less visible for the programmer and relying more on a sophisticated runtime system. Future results following this new approach can be compared to the more traditional one followed in this project.

References

[1] M.P. Allen and D.J. Tildesley. *Computer Simulation of Liquids.* Oxford Science Publications, 1987.

[2] K. Esselink and P.A.J. Hilbers. Parallel molecular dynamics on a torus network. In *Scalable High Performance Computing Conference SHPCC '92*, pages 106–112. IEEE Computer Society, IEEE Computer Society Press, April 26–29 1992.

[3] K. Esselink, B. Smit, and P.A.J. Hilbers. Efficient parallel implementation of molecular dynamics on a toroidal network, Part I: Parallelizing strategy. *J. Comp. Phys.*, 1992. accepted.

[4] J.-P. Hansen and I.R. McDonald. *Theory of Simple Liquids.* Academic Press, London, 1990.

[5] Johan J. Lukkien and Jan L.A. van de Snepscheut. A tutorial introduction to Mosaic Pascal. Caltech-CS-TR 91-02, California Institute of Technology, 1991.

[6] Steve Plimpton and Grant Heffelfinger. Scalable parallel molecular dynamics on MIMD supercomputers. In *Scalable High Performance Computing Conference SHPCC '92*, pages 246–251. IEEE Computer Society, IEEE Computer Society Press, April 26–29 1992.

[7] D.C. Rapaport. Multi-million particle molecular dynamics II. Design considerations for distributed processing. *Comp. Phys. Comm.*, 62:217–228, 1991.

[8] Semiannual Technical Report. Darpa submicron systems architecture project. Caltech-CS-TR 91-10, California Institute of Technology, November 1991.

[9] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Caltech-CS-TR 88-1, California Institute of Technology, 1989.

[10] Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys. Rev.*, 159(1):98–103, 1967.