HAM: Hardware Moved Molecules
Annual Report 1997

Martin Gerber[1], **Thomas Gössi**[2]

[1]Computer Engineering and Communication Networks Lab (TIK), gerber@tik.ee.ethz.ch
[2]Electronics Laboratory (IFE), goessi@ife.ee.ethz.ch

Swiss Federal Institute of Technology (ETH), Gloriastr. 35, CH-8092 Zürich

ABSTRACT

In this report the work of the first 18 months of the Polyproject “Parallel Computing in Quantum and Classical Molecular Dynamical Simulation“ is summarized. The project is planned for a total duration of three years. Four labs of the ETH attend the project: IGC (Computational Chemistry Group, head of the project), IFE (Electronics Lab), TIK (Computer Engineering and Networks Lab) and IWR (Scientific Computing Lab). GROMOS (GROningen MOlecular Simulation package) is a computer simulation tool which is distributed and supported in various versions by the Computational Chemistry Group. The main goal of the project is to accelerate the simulation of molecules in liquids by a factor of ten. To achieve the goal dedicated hardware must be developed to speed-up the Gromos software. The main topics in this report are as follows: First we present profiling results of the Gromos program, performed on different workstations and processors. Then we give an overview on existing hardware accelerators. New modelling techniques for the molecular dynamics algorithm are presented as well as models for different new parallel hardware solutions. With these models a design space exploration was performed using techniques such as system synthesis using Evolutionary Algorithms.

Contents

1 Introduction

Since 1978 W. van Gunsteren has developed a set of programs for computer simulation of biomolecular systems. This set is called GROMOS (GROningen MOlecular Simulation package). New and improved simulation methodologies are continuously developed by his research group, which are then implemented in Gromos. The Gromos software is currently used by more than 400 academic and industrial research groups all over the world [23].

The simulation of a biomolecular system aims at the interaction between proteins and a solvent, which is pure water in most cases. The molecular simulation can iteratively be calculated as follows: The distances between all atoms are calculated, then forces between molecules, their energy and the overall pressure are calculated from the distances and at last the new molecule-positions from the forces. These steps are repeated until the simulation terminates.

The computation can be simplified by not having to calculate the forces between molecules, whose distances exceed a certain value and therefore do not significantly interact. Thus Gromos generates a so-called “pairlist“, where those molecules are entered which interact. Given the fact, that positions only change little every iteration-step, the pairlist has only to be updated every 5th to 10th iteration. With this, calculations can drastically be reduced.

For a biomolecular system simulation, the molecules can be separated into two groups: solute (protein) and solvent molecules. Basically every molecule of each group may interact with any molecule of its own or of the another group. Therefore, three cases have to be distinguished: distance and force calculations *a)* between solvent molecules, *b)* between solute molecules and *c)* between solute-solvent molecules. Most of the systems consist of much more solvent molecules than solute molecules. As shown by analysing some benchmark programs, most of the calculation time is used for the solvent-solvent part, where about 100 millions floating-point operations per iteration step are required [16].

Due to the considerably large number of particles, molecular simulations need a lot of computer performance. Thus Gromos has already been implemented on networks of Transputers and parallel Supercomputers as MUSIC or ALPHA 7 [2]. But it turned out that this computers are too expensive and not the right choice for molecular dynamics simulation. In addition workstation clusters are not suitable to solve this problem since they lack the communication bandwidth to timely exchange data between iterations. The best choice is a coprocessor which is especially built to compute the Gromos algorithm and which can simply be linked up to commercial workstations. A projected speed-up of ten for Gromos can be achieved with such a coprocessor compared to a single, state of the art workstation.

An interdisciplinary Polyproject was started in 1996, to develop an appropriate coprocessor and embed it into the Gromos environment. The project is called “Parallel Computing in quantum and classical molecular dynamical simulation“. Classical molecular dynamics includes the solving of Newton’s classical equation of motion. In the quantum dynamics simulation the Schrödinger equation is considered and Eigenvalues has to be determined. The following labs participate the project:

- IGC (Computational Chemistry Group), responsible for algorithms and chemical models. This lab is the head of the project and provides the Gromos software in various versions for different platforms (including some parallel machines).

- IWR (Scientific Computing), responsible for analysis and accelerating of algorithms concerning the quantum part of the simulation. To solve the Schrödinger-equation, fast parallel Eigenvalue solvers must be developed and implemented.
- TIK (Computer Engineering and Networks Lab), responsible for computer and software engineering, state of the art system design, and design methodology.
- IFE (Electronics Lab), responsible for implementation of the hardware and general hardware aspects.

The architecture of the coprocessor will consist of one or more DSP- or RISC-processors in combination with FPGAs (Field Programmable Gate Array). FPGAs will be used for simple calculations and the communication handling whereas the processors will be used to calculate the more complicate algebraic expressions. Eventually an ASIC or an MCM must be developed.

The responsibilities within the project led to a close co-operation of the labs TIK and IFE in the initial phase of the project. Because of that, this report is a summary of the work of both labs.

A general molecular dynamics introduction is presented in paragraph 2, including the basic principles in computational physics and chemical models of interaction potentials. In paragraph 3 the initial Gromos software has been analysed: The Gromos force field is described as well as some benchmarks and profiling results. Models of the most time consuming functions of Gromos are developed for further use in the design space exploration sections. Paragraph 4 gives an overview on existing hardware accelerators. New hardware architectures and their models are presented in paragraph 5. In paragraph 6 the performance of the Sharc DSP is measured with a typical molecular force calculation function. In addition, the distance calculation routine was implemented on a FPGA to test the complexity and usability of the today's most complex gate arrays for our application. Paragraph 7 summarises how the specification of the MD algorithm was implemented in Mathematica and describes a system synthesis technique using genetic algorithms to perform design space exploration semiautomatically. Last, some conclusions and the further work is outlined in paragraph 8.

2 Molecular Dynamics Simulation Methods

2.1 Survey

Computer simulations are a common tool to investigate dynamic, thermal and thermodynamic properties in molecular systems. Two major methods have been developed in the recent years: The Monte Carlo method (MC), so called because of the role that random numbers play in the method, and molecular dynamics (MD).

In an MC calculation a Markov chain of particle configurations representing the micro-state of the observed system is constructed as follows: From a given initial state a transition to the next state is performed by a randomly chosen particle is moved along an also randomly chosen vector. On the basis of the total energy of the system the Metropolis algorithm is used to decide whether the new state is added to the chain or the old state counts again. This Markov chain converges to the thermodynamic equilibrium for some million states. Structural and thermodynamic properties of the investigated substance can easily be obtained. It is also possible to simulate non-equilibrium processes, with a dynamic interpretation of the random movement of the particles (random walk).

The Molecular Dynamics (MD) method is based on numerical integration of Newton's equation of motion. Each time step, the interaction forces between the involved particles in a simulation box must be determined. After the integration step the velocities of the particles are known and the new positions can be calculated for the next step. The result is a spatial trajectory for each particle. With this data and using statistical physics laws, all interesting structural, thermal and thermodynamic properties can be evaluated. Typically, a time step is two picoseconds and a run over about 100 steps is performed. For very complicated systems with a lot of intra- and intermolecular interactions the simulation is very time consuming.

The focus in this report is on MD Simulations with a lot of particles or atoms (20,000 and more). Smaller systems are fast and easy to simulate with modern workstations with more than 10 SPECfp95 Mflops peak performance, but large systems with 20,000 and more particles require a lot of time even on the today fastest workstations. Therefore the demand to acceleration techniques (new algorithms, special purpose hardware) is big, as we will see in the next sections.

As mentioned before, we have to solve the equations of motion. To get the velocities, we integrate the forces of each particle. With the width of one timestep the new positions are updated. The art of a good molecular dynamics simulation run is the right choice of a force field: the totality of all interactions and the numerical models. The integration itself consumes relatively small time in the overall simulation as we will see in the profiling section. So, the integration algorithm can be chosen very accurate without losing performance. To get an overview over the huge amount of different techniques we give a short introduction to the physico-chemical basics of atomic interaction and the corresponding numerical algorithm proposals in the literature.

2.2 Model Systems and Interaction Potential

2.2.1 N-Body systems

We assume an atomic system with N particles, where it does not matter if the N particles belongs to M molecules or not. The notation of the sums in eqn (2.1) indicates a summation over all distinct pairs i and j without counting any pair twice. The same care must be taken for triplets etc. The potential energy V may be divided into terms depending on the coordinates of the involved particles, pairs, triplets, etc.:

$$V = \sum_i v_1(\mathbf{r}_i) + \sum_i \sum_{j>i} v_2(\mathbf{r}_i, \mathbf{r}_j) + \sum_i \sum_{j>i} \sum_{k>j>i} v_3(\mathbf{r}_i, \mathbf{r}_j, \mathbf{r}_k) + \ldots \tag{2.1}$$

The first term v_1 in eqn (2.1) represent the external field of the system. The other terms represent the particle interactions. It is obvious that for one particle all other particles deliver a more or less important portion of the interaction potential. In practice all terms in the order higher than three can be neglected, but the three body potential unfortunately is not.

The basic equation of motion for quantum mechanics is the time dependent Schrödinger equation. In classical simulations, the Hamiltonian H of the Schrödinger equation has the form

$$H(\mathbf{p}, \mathbf{r}, m, s) = K(\mathbf{p}, m) + V(\mathbf{r}, s) = \sum_{i=1}^{N} \frac{1}{2} \cdot m_i \mathbf{v}_i^2 + V(\mathbf{r}, s), \tag{2.2}$$

where the first term is the kinetic energy term depending on the momenta and mass. The second term is the potential energy or interaction function describing the interaction energy in terms of particle coordinates and force field parameters. The force f_i on particle i due to a particular interaction term is given by the relation

$$\mathbf{f}_i = -\frac{\partial}{\partial \mathbf{r}_i} V(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) \tag{2.3}$$

The MD trajectories depend on the forces on the atoms, not on the energies. In the next paragraphs, we describe the forces and algorithms to calculate trajectories in more detail.

2.2.2 The Potential Model

The potential in eqn (2.3) is composed of nonbonded interactions (van der Waals and electrostatic interaction) and interactions between covalently bonded atoms (van der Waals Dispersion, Coulomb force). In a general approach, we model the potential as follows:

$$V^{tot} = V^{bond} + V^{angle} + V^{tors} + V^{nonb} + V^{H} \tag{2.4}$$

The total potential in a molecular system V^{tot} is composed of bond-stretching, bond-angle, torsion-angle, nonbonded and hydrogen bonds sub-potentials. One of the main tasks in a Molecular Dynamics simulation is the determination of specific models for these potentials.

2.2.3 Pair Potentials

At the simplest level, the interaction occurs between pairs of atoms and is responsible for providing the two principal features of an interatomic (or intermolecular) force: The first is a repulsive part for small distances or the resistance to compression due to the nonbonded overlap between the electron clouds, the second is the attractive part due to the binding of the atoms (van der Waals dispersion). Additional Coulomb terms appear for charged species.

Molecules are represented by atoms with orientation-dependent forces, or as structures containing several interaction sites. If the molecules are rigid, flexible, or somewhat between, and if there are internal degrees of freedom, there will be internal forces leading to the separated potential proposed in eqn (2.4). Staying at the simple level with only nonbonded interactions without covalent bonds, a model for the pair potential and for the non-additive three body potential is required.

One of the best known pair potentials was initially proposed for liquid argon and was derived by considering a huge quantity of experimental data. At liquid densities, where the three body potential is significant, also a lot of estimates has been made of the leading three-body contribution. These experiments show that about 10 per cent of the energy may be from the non-additive potential terms, e.g. the terms of the order three and higher. Because the calculation of sums over triplets of particles is very expensive in the sense of computational cost, they are only rarely included in computer simulations. Fortunately the model of the pairwise potential can be modified to partially include the many-body effects by defining a new “effective“ pair potential:

$$V \approx \sum_i v_1(\mathbf{r}_i) + \sum_i \sum_{j>i} v_2^{eff}(r_{ij}) \tag{2.5}$$

Almost all pair potentials in computer simulation are regarded as effective pair potentials so it is convenient to write only $V(r_{ij})$. The most widely used approximation to the effective pair potential is the Lennard Jones (LJ) potential. Assuming an atom pair i and j the LJ-potential energy is as in eqn (2.6), depending only on the magnitude of the pair separation r_{ij}.

$$V^{LJ}(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{2.6}$$

If the parameters are chosen appropriately, eqn (2.6) provides a reasonable description of the properties (e.g. argon or other one-atomic liquids). Assuming only two atoms, the approximation is wrong, of course, because the many-body correction is included in the parameters.

The solid line in fig. 2.1 shows the pair potential for a liquid atom pair (experimental, measured data), the dashed line illustrates the corresponding 12-6 LJ approximation for systems with lots of atoms and therefore including many-body interactions.

Sometimes even simpler pair potentials are used to perform very fast simulation runs, e.g. hard-sphere potentials and square-well potentials.

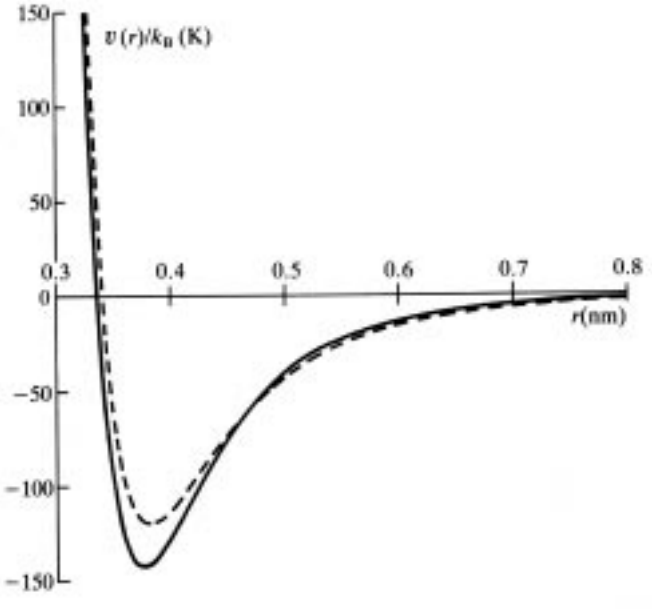


Fig 2.1 Liquid argon pair potential

For ions it is necessary to add the Coulomb charge-charge interaction eqn (2.7) to the pair potential.

$$V^{qq}(r_{ij}) = \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \tag{2.7}$$

For ionic systems, induction interactions are important: the charge induces a dipole on neighbour ions. This term is not pairwise additive and hence it is necessary to introduce more correction terms.

The potentials for covalently bond forces depend on the model of the molecule: Which bonds have which degrees of freedom, where are fixed bond lengths etc. Generally a special model for each molecule is needed. A good computer simulation program supports all known intramolecular interaction models. The choice of an adequate model may depend on several parameters: The temperature, the involved atoms, hydrogen bonds, constraints, etc. Refer to [18], [26] for more details.

A lot of research has already been done in finding fast (parallelizable) algorithms for these models. Because most of the computation time of MD simulation with liquids is spent to calculate the nonbonded forces, the acceleration potential for a whole run is small even if the bonded forces computation gets significantly faster. We must speed up the calculation of the nonbonded forces. We now discuss the general MD algorithm, and, in the next paragraph, the Gromos specific force field which slightly varies from the model described above.

2.3 The Universal MD Algorithm

The Verlet algorithm is perhaps the most widely used method of integrating the equations of motion. The overall scheme is illustrated in fig. 2.2. The method is based on the positions and acceleration of the particles of the current time step and the positions of the last step.

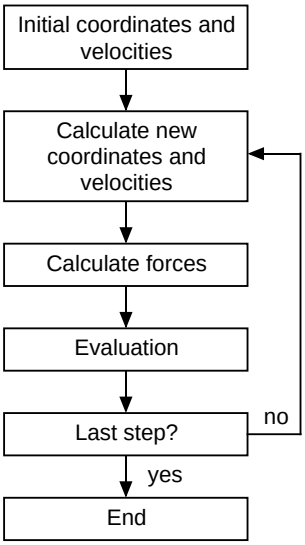


Fig 2.2 Universal MD algorithm

The equation for advancing the positions reads as follows:

$$\mathbf{r}(t + \delta t) = 2 \cdot \mathbf{r}(t) - \mathbf{r}(t - \delta t) + \delta t^2 \mathbf{a}(t) \tag{2.8}$$

Note that the velocities do not appear in this equation. They have been eliminated by adding the Taylor expansions of $\mathbf{r}(t + \delta t)$ and $\mathbf{r}(t - \delta t)$:

$$\mathbf{r}(t + \delta t) = \mathbf{r}(t) + \delta t \mathbf{v}(t) + \frac{1}{2}\delta t^2 \mathbf{a}(t) + \ldots \tag{2.9}$$

$$\mathbf{r}(t - \delta t) = \mathbf{r}(t) - \delta t \mathbf{v}(t) + \frac{1}{2}\delta t^2 \mathbf{a}(t) - \ldots \tag{2.10}$$

The velocities are not needed to compute the trajectories, but are useful to calculate the kinetic energy. To get the velocities, simply subtract eqn (2.10) from eqn (2.9).

There are several other algorithms to calculate the trajectories, the most often used are the velocity-version of the Verlet algorithm, the Beeman algorithm and Gear's algorithm, all of them well documented in the literature. A popular integration method is the leap frog algorithm [18], [19], [20].

3 Gromos Analysis

3.1 The Gromos Force Field

The Gromos force field is defined as

$$V^{phys} = V^{bon}(\mathbf{r}, s) + V^{nonb}(\mathbf{r}, s) \tag{3.1}$$

$$V^{bon}(\mathbf{r}, s) = V^{bond}(\mathbf{r}, s) + V^{angle}(\mathbf{r}, s) + V^{har}(\mathbf{r}, s) + V^{trig}(\mathbf{r}, s) \tag{3.2}$$

following the potential described in eqn (2.4) and eqn (2.5). Within the Gromos package, it is also possible to define special non-physical interactions for atomic positions restraining, distance restraining and more. For further details refer to [23], here we mention only the standard physical forces which separates into bonded and nonbonded forces. We give a short description of the partial forces for covalently bonded atoms, for formulas refer to [23] or the literature ([18], [19], [20], [26]).

bond: covalent bond-stretching interaction. This force generally has to be evaluated for all covalent bonds in the molecular system.

angle: covalent bond-angle bending interaction. This force generally has to be evaluated for all covalent bond angles in the molecular system.

har: harmonic dihedral-angle bending interactions can be used to keep groups of atoms in a special spatial arrangement. For example a planar atomic ring configuration can be suppressed to deviate from the plane.

trig: trigonometric dihedral-angle torsion interaction. See [18], [23].

nonb: nonbonded (van der Waals and electrostatic) interaction

$$V^{nonb} = \sum\left\{\left[\frac{C_{12}(i,j)}{(r_{ij})^6} - C_6(i,j)\right]\cdot\frac{1}{(r_{ij})^6} + \frac{q_i q_j}{4\pi\varepsilon_0\varepsilon_1}\cdot\left[\frac{1}{r_{ij}} - \frac{C_{rf}(r_{ij})^2}{2(R_{rf})^3} - \frac{2-C_{rf}}{2R_{rf}}\right]\right\} \tag{3.3}$$

The sum in eqn (3.3) is over all distinct nonbonded pairs ij. The force on one atom pair i,j due to eqn (3.3) is

$$\mathbf{f}_i = \left[\frac{2C_{12}(i,j)}{(r_{ij})^6} - C_6(i,j)\right]\cdot\frac{6\mathbf{r}_{ij}}{(r_{ij})^8} + \frac{q_i q_j}{4\pi\varepsilon_0\varepsilon_1}\cdot\left[\frac{\mathbf{r}_{ij}}{(r_{ij})^3} + \frac{C_{rf}\mathbf{r}_{ij}}{(R_{rf})^3}\right] \tag{3.4}$$

with

$$\mathbf{f}_i = -\mathbf{f}_j \tag{3.5}$$

Compared with the proposed force field in paragraph 2.2 the Gromos force field does not contain a special term for hydrogen bonds. Instead, the van der Waals repulsion will be increased for hydrogen bonds.

3.2 MD Algorithm in Gromos

The integration of the equation of motion is done with the leapfrog algorithm. Generally, a simulation may include several options like weak temperature coupling, a pressure bath in a box, periodic boundary conditions, distance and other constraints, virial calculation and more. The output files, actually the trajectories for positions, velocity and energies etc., are also user specified and may be but must not be written.

There are some well-known models to describe this kind of algorithm, e.g. flowcharts describing sequence or control flow. Other models like data flow graph may not represent any control flow. Heterogeneous models like control/data flow graph or structure charts are weak coupled data flow and control flow graphs or are unsuited for a distributed target implementation. What we need is a specification model supporting several control structures like data dependent branches and loops and the possibility to specify data dependencies and parallelism within one single graph.

3.2.1 Sequence Graphs and Data Flow Graphs

We tried to model the MD algorithm with a pure data flow graph. This can easily be done but is not very flexible for functions which need not being evaluated each time step or to parallelize functions generically. Since the algorithm in principle is data flow and control flow driven, which means how the simulation proceeds is dependent on the availability of data structures like coordinates after the integration step but also on the choice of the options described above, the desired graph is a dynamic data flow graph. For this reason we developed a new formalism deduced from standard data flow models ([17], [27]).

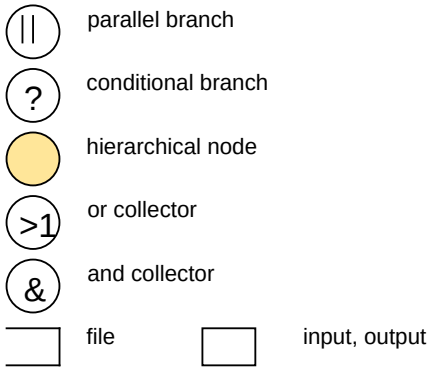


Fig 3.1 Special nodes in the HaMM data flow

Our graphical representation of the control/dataflow graph (CDFG) uses rectangles for input or output nodes (io node), circles for activity nodes, and open-ended rectangles for data store nodes. Data flow is represented by arcs, labelled with the associated data. The *parallel branch* has a data distribution functionality: One input edge and as many as you like output edges are allowed for this node. The input data is routed selectively to the target nodes and all target nodes are triggered to execute in parallel (if possible). The *conditional branch* activity requires a conditional expression associated to each of the output edges in addition to the data. The condition is true for exactly one subsequent node which is triggered to execute with the respective input data subset. Data distribution is performed in the same way as for the parallel branch, one input edge and many output edges are allowed. A *hierarchical node* may have several inputs and outputs and is just a graphic element to improve the overall view. The *or collector* accepts many inputs and has one output: If new and valid data is

present on at least one input, the output and the next node is triggered with the data derived from this first valid input. Care must be taken to keep the data dependencies compatible. The *and collector* also accepts many inputs and has one output, but all inputs must be valid to trigger the next activity. The output data is the combination of all inputs and the input data must be distinct. A *file* is a data sink and can have many inputs, an *io node* can produce and consume data, also many edges are possible. All nodes described above may not have any functional implementation. The *input/output* node is the only one where arrows in both directions are possible. In addition to these special control nodes of course there are data processing nodes with an algorithmic functional implementation.

As an example, we describe parts of the Gromos MD algorithm with this rules. In a first step we described the main MD simulation loop of the Gromos simulation package with this method. In fig. 3.2 the top of the hierarchy for an MD run is shown, on the left the call of the main MD loop for each time step and the files to be read and written, on the right the forces calculation and integration procedure with optional virial calculation.

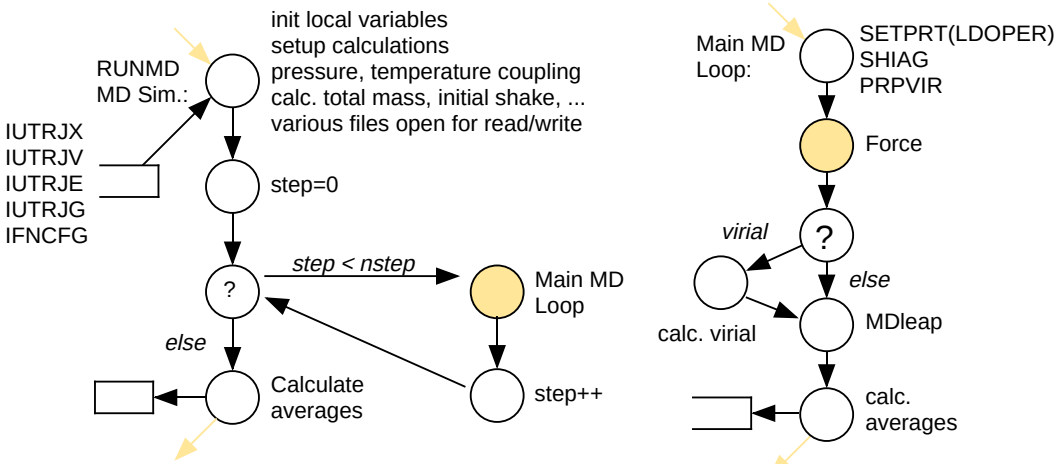


Fig 3.2 Gromos MD algorithm

The next steps include the refinement of the structure. Fig. 3.3 shows the loop over atoms to calculate nonbonded interactions in C programming language and in a CDFG representation without the data on the edges. Expressions for the conditional branch are written in italic, the statements for the activity nodes are given directly or as a procedure or function name. Note that the diagrams in fig. 3.3 and fig. 3.2 are heavily simplified.

Now we have a powerful formal method to describe the MD algorithm in a flexible but efficient and simple way. The sequential program structure can be represented in a way allowing parallel and/or sequential implementation. The further work included the refinement of the functions: In the original Gromos version the nonbonded forces calculation is one function consuming 50 per cent or more of the overall simulation time. For this reason it is necessary to split the nonbonded forces calculation routine into subfunctions according to fig. 3.4 and to state a generic parallel model for at least the solvent-solvent part of the nonbonded interaction with the new specification method. For this task it may be useful to introduce new activities to describe algorithmic parallelism.

The major benefit is a formal representation of the algorithm which is easy to change: Introducing new functions in the graph without modifying or knowing anything about the source code is possible.

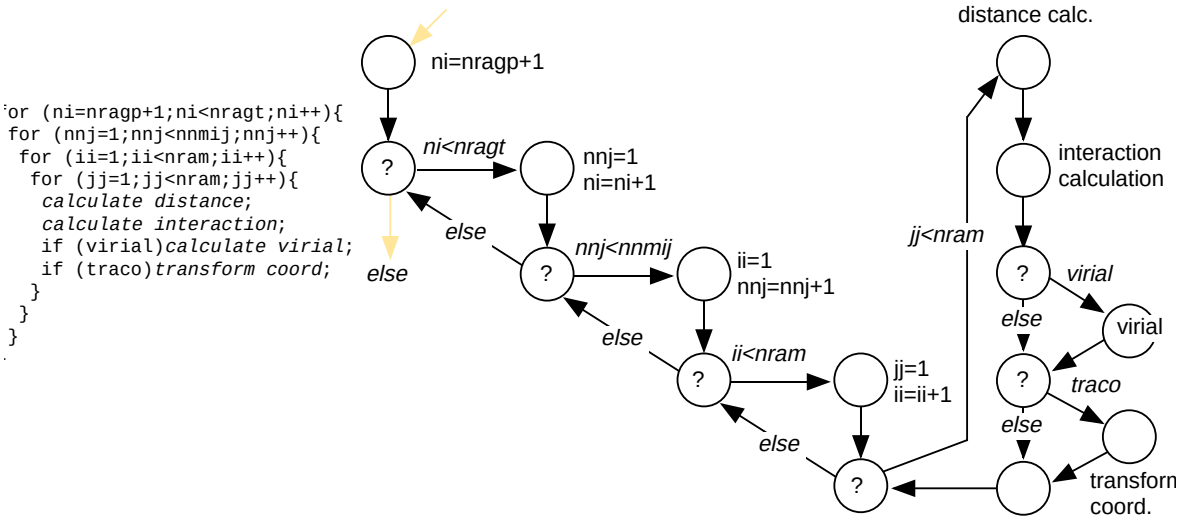


Fig 3.3 Nonbonded forces calculation

3.2.2 Searching Neighbours, long-range Interaction

The nonbonded interaction between atoms decrease with the distance between them and generally in MD simulations only interactions between atoms with a distance lower than a certain cutoff distance are taken into account. Unfortunately the coulombic interaction is a so called long-range interaction due to the 1/r distance dependence, and a typical cutoff distance like 2.0nm - 3.0nm is required for adequate accuracy. Because the number of neighbours for nonbonded interactions grows with the third power of the cutoff radius, the computation time for a 3nm cutoff is not acceptable for efficient simulation. To overcome this problem of long-range forces lattice methods like the Ewald sum or reaction field methods can be applied. In the reaction field method the field consists of two parts: A short-range contribution from neighbouring molecules within a cutoff sphere and a long-range part from molecules outside the cutoff sphere which is considered to form a dielectric continuum. Together with the charge groups concept, the cutoff radius of the short-range part decrease from 3nm to about 1nm - 1.5nm. Within a charge group, the sum of partial charges of the atoms adds up to exactly zero. Therefore the electrostatic interaction between two such groups is of dipolar character and decreases with the third power of the distance. This leads to some restrictions: Charge groups may not be split over simulation box boundaries and the interaction is calculated either for all atoms of a charge group to all atoms of a neighbouring charge group or for none.

Other approaches to handle long-range forces are the already mentioned Ewald sum ([18] and [20]), the P^3M method (particle-particle, particle-mesh) [19], Multipole expansion schemes due to Ladd (1977/78) ([19], [18]) or the Greengard-Rokhlin algorithm. In the P^3M method the short-range part of the potential is handled normally, the long-range part is calculated using a particle mesh technique (e.g. solving the Poisson's equation using FFT).

In Gromos, a solvent molecule is always one charge group, the solute molecules are composed of many charge groups. The simplest way to find the neighbouring charge groups of a primary charge group is to scan over all possible pairs in the system. If the system contains N charge groups the number of pairs to consider is $1/2 \cdot N^2$. Constructing the pair list in this way requires a lot of the overall simulation time, although the pairlist must not be constructed every time step. The update interval to calculate the pairlist is user-specified and usually lies between three and five. In the present implementation, two cutoff spheres are

available where the smaller is used to build the pairlist. For the particles between the two spheres the interaction is calculated during the pairlist construction and is kept constant until the next pairlist build. This special interaction is called long-range or twin-range interaction.

To keep the distortion small at cutoff sphere boundaries or rather to treat the long-range part, a Poisson-Boltzmann reaction field force correction is applied.

3.2.3 Nonbonded Interactions, Periodic Boundaries

The nonbonded forces calculation distinguishes due to the different interaction potentials into solute-solute, solute-solvent and solvent-solvent interactions. The partitioning into these parts depends on the solvent atoms to solute atoms ratio, see fig. 3.4.

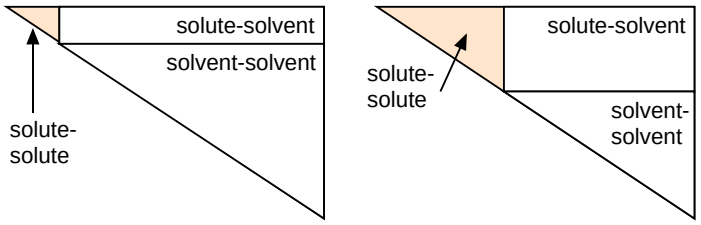


Fig 3.4 Nonbonded forces decomposition

To reduce computation time for proteins in liquids the number of solvent atoms can be reduced using periodic boundary conditions, also, surface effects can be avoided using this technique. With periodicity, the rectangular box is replicated throughout space to form an infinite lattice. If a molecule moves in the original box, its periodic image moves exactly in the same way in the neighbouring boxes. If a molecule leaves the central box, one of its images will enter through the opposite face. In Gromos package, a set of different central box geometries can be periodically repeated: rectangular, truncated octahedron (a octahedron fitted in a rectangle) and monoclinic (rectangular box with oblique angles) boxes.

3.3 Gromos Benchmarks and Profiling

As a benchmark we have chosen a system similar to the main problem we want to address: A solute, typically a protein, surrounded by solvent in periodic boundary conditions.

	with water, octah.	**with water, rect.**	**without water, octah.**
Number of solute atoms	3,078	3,078	3,078
Number of solvent atoms	16,281	32,883	0
Total number of atoms	19,359	35,961	3,078
number of solute charge groups	1,285	1,285	1,285
Number of solvent charge groups	5,427	10,961	0
Total number of charge groups	6,712	12,246	1,285

Table 3.1 Benchmark systems

We present the results for octahedron and rectangular boundary conditions with water and the solute without water. The system sizes are given in table 3.1, where the protein involved is Thrombin. For all measurements 100 steps are simulated. If nothing else is specified, the machine for all simulations in this paper is a 170MHz Sun Ultra 1 Creator workstation with 64MB RAM running under solaris 2.5.1.

The first example is a simulation with water in a truncated octahedron. Every 5 time steps the pairlist was updated with a 1.4nm cutoff without long-range interaction. The output for a typical run generated with gprof is summarized in table 3.2.

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 2096.96
seconds
 %  cumulative    self              self     total
 time    seconds  seconds    calls  ms/call  ms/call  name
 83.0    1739.92  1739.92      100 17399.18 17399.18  nonbml_ (24)
 15.5    2065.24   325.32       20 16266.07 16266.07  nbnone_ (31)
  0.9    2084.55    19.31      204    94.66    94.66  shake_ (17)
  0.1    2087.06     2.51      100    25.10    25.10  mdleap_ (23)
  0.1    2088.98     1.92      400     4.80     4.80  dihang_ (16)
  0.1    2090.52     1.54      200     7.70     7.70  angle_ (18)
  0.0    2093.25     0.38      100     3.80     3.80  shiag_ (25)
  0.0    2093.62     0.37       12    30.83    30.83  cenmas_ (35)
  0.0    2094.90     0.21   207600     0.00     0.00  chpstr_ (2)
  0.0    2095.24     0.15    38718     0.00     0.00  chpnre_ (7)
  0.0    2095.77     0.12      100     1.20     1.20  force_ (22)
  0.0    2095.87     0.10     1242     0.08     0.08  _times (502)
  0.0    2096.13     0.07    57283     0.00     0.00  gimme_ (5)
  0.0    2096.24     0.05   165200     0.00     0.00  getdmp_ (3)
  0.0    2096.29     0.05    77828     0.00     0.00  chpint_ (4)
  0.0    2096.33     0.04    12393     0.00     0.00  chkmty_ (9)
  0.0    2096.37     0.04       20     2.00     2.00  nbpml_ (32)
  0.0    2096.41     0.04        2    20.00    20.00  getarr_ (57)
  0.0    2096.45     0.04        1    40.00    40.00  gtcoor_ (101)
  0.0    2096.49     0.04        1    40.00    40.00  runmd_ (133)
  0.0    2096.59     0.03    44874     0.00     0.00  lismty_ (6)
  0.0    2096.68     0.03       20     1.50     1.50  clcgeo_ (30)
```

Table 3.2 A typical gprof output. Pairlist every 5 steps, cutoff 1.4nm, octahedron boundaries

Subroutine *nbnone* calculates the pairlist without long-range interaction, *nonbml* calculates the nonbonded forces, the *shake* routine handles constraints, *leap* is the integration step and *times* is an extra timing routine introduced to test the gprof results and allowing measuring times within a subroutine. The times derived from this function are listed in table 3.3.

	water, octah.	**water, rect.**	**no water**
Main MD loop	213,038	369,326	18,336
force	210,787	361,821	18,027
Integration (leap)	2,175	4,107	303
nbnone (Pairlist)	32,544	88,165	669
nonbonded (all)	177,745	273,122	16,895
nonb. solute-solute, solute-solvent	51,645	45,700	16,892
nonb. solvent-solvent	126,094	227,419	0

Table 3.3 Timing measurement with C routine times (values in 1/100 sec.)

It is obvious that most of the simulation time is spent in constructing the pairlist and calculating nonbonded interactions. Except for *shake*, all other tasks do not really matter. The function *times* allows us to set and stop timers from anywhere in the application. The numerical values in the left column of table 3.3 may be compared with those in table 3.2. The results for the other benchmarks are also listed in table 3.3, with the same pairlist update intervals and cutoff.

3.4 Gromos Functions Modelling

The number of floating point operations and the number of fixed point operations in dependence on the problem parameters will be a good unit of measurement of function complexity. Of course only subroutines with lots of floating point operations can be modelled in this way. Fortunately the pairlist construction and nonbonded forces calculation use practically nothing else as floating point operations in the inner loops. With the number of operations, the calculation time and the performance specifications of the workstation (benchmark, Mflops, Mops) a unit of measurement is available to scale the computation time in dependence of the host performance and the simulation problem complexity. Other important function parameters are the amount of input and output data, separated into static and dynamic data.

3.4.1 Solvent-solvent Non-bonded Interaction

In table 3.4 the number of floating point operations are shown for one atom-atom interaction of the solvent-solvent part of the nonbonded forces calculation. It is assumed that periodic boundary conditions are always applied, for systems without periodic boundaries the first column of table 3.4 may be taken, with P1 equals zero.

Op.	**locto = false** **ldotra = false** **ldovir = false**	**locto = true** **ldotra = false** **ldovir = false**	**locto = false** **ldotra = true** **ldovir = false**	**locto = true** **ldotra = false** **ldovir = true**	**locto = false** **ldotra = true** **ldovir = true**
+, -	4*ndim +1(4D) +1*(ndim*P1) +14	4*ndim +1(4D) +1*(ndim*P1) +5*(ndim*P2) +17	4*ndim +1(4D) +1*(ndim*P1) +11	4*ndim +1(4D) +1*(ndim*P1) +5*(ndim*P2) +26	4*ndim +1(4D) +1*(ndim*P1) +25
*	2*ndim +1(4D) +15	2*ndim +1(4D) +2*(ndim*P2) +15	2*ndim +1(4D) +19	2*ndim +1(4D) +2*(ndim*P2) +21	2*ndim +1(4D) +25
/	1	1	1 + 2[a]	1	1 + 2[a]
	1	1	1	1	1
sin	0	0	1[a]	0	1[a]
cos	0	0	1[a]	0	1[a]

Table 3.4 Atom-atom interaction: FP Operations per time step

a. Subroutine TRACO, coordinate transformation

With

ndim the number of dimensions to calculate in (three or four),

4D equals zero for three dimensional simulations, equals one for four dimensional simulation,

P1 the probability for periodic boundary correction: One out of three coordinates is not in the periodic box (rectangular and octahedron), therefore a correction is necessary,

P2 which is like P1, but additional for octahedron box,

locto which is true if periodicity is applied with a truncated octahedron (beta=90) box,

ldotra indicating a monoclinic box (beta not 90) if it is true, and

ldovir enabling the virial calculation if it is true.

The number of floating point operations for one time step reads as

$$OPS_1 = OPS_{1aa} \cdot CG_{solv} \cdot NRAM^2 \cdot NPC, \tag{3.6}$$

with

OPS_1 the number of operations in one time step (solvent-solvent interaction only),

OPS_{1aa} the number of operations for one atom-atom interaction (distance and interaction), derived from table 3.4,

CG_{solv} the total number of solvent charge groups (molecules),

NRAM the number of atoms per solvent molecule, and

NPC the average number of (solvent) neighbours per solvent charge group.

The most insecure parameter is NPC. There is only one possibility to get an exact value for NPC for a given cutoff: Run a simulation and calculate the average. It would be easy to give a formula for the average number of pairlist entries in function of the cutoff radius, but it is not possible to predict the number of pairlist entries for the solvent-solvent part, because of the density fluctuations in the solvent near the protein. The next paragraph concern the input and output data of the solvent-solvent interaction function.

Dynamic data is defined as data that change at least once in a simulation run or rather within the main MD loop. By contrast a dynamic data element must not change every time step, but has an additional parameter specifying how often the value is updated per occurrence. The pairlist for example is calculated every three time steps, so the overall data transfer rate for this function is reduced by this factor.

The main data transfer for solvent-solvent force function is composed of the input coordinates of the particles, the respective pair list if there is a new one, and the partial forces on the atoms as output. Other parameters like the length of the simulation box does not matter. In the following, we give a parametrised approximation for the dynamic data amount to be transferred each time step if there is one force processor in the system. In paragraph 7.1 we give a generic model for many processors.

$$M_{solvsolv} = ndim \cdot (2 \cdot nattot - nrp) + CG_{solv} + NPC \tag{3.7}$$

Relevant are the array of the input coordinates XCOORD[ndim (nattot-nrp)] of the solvent, the force array F[ndim nattot], the pairlist pointer list INB[CG_{solv}] and the pairlist JNB[NPC] itself. Many other dynamic input and output parameters and data do not matter for systems with more than several hundred solvent atoms. The parameters in eqn (3.7) are:

nattot, the total number of atoms,

nrp, the number of atoms per solute molecule. Within this report we always consider the same example with one solute molecule.

We assume that integer and floating point numbers are one word (4 Bytes). So M is in unit words.

3.4.2 The Pair List Concept

The pairlist construction can be separated into two tasks: The calculation of the centres of the charge groups and the calculation of the pairlist itself. In table 3.5, first column, the floating point operations for the calculation of the centres are outlined. Note that for the solvent molecules the first atom is assumed to be the heaviest and therefore approximating the centre. This leads to the situation that in this routine also a lot of copy operations and fixed point operations occur, but the number of floating point operations is still dominant.

The actual Gromos software is delivered with a slow pairlist algorithm, because all possible pairs are scanned. Therefore, the distance calculation is executed exactly

$$n = \frac{NRAGT \cdot (NRAGT + 1)}{2} \tag{3.8}$$

times. Except for the distance calculation, there are no other floating point operations in the pairlist subroutine. In addition to the ordinary pairlist construction the user may specify a twin-range. If this is the case, the interaction related to the twin range is evaluated during the pairlist construction. In our benchmark we do not include twin-range, as well it is better to simulate with a larger cutoff than with a twin-range. The number of floating point operations for distance calculation for different periodic boundary conditions are summarised in table 3.5.

Op.	**Geom. centre**	**rectangular**	**octahedron**	**monoclinic**
+/-	$\left(\frac{NRP}{NCAG} - 1\right) \cdot NDIM \cdot NCAG \cdot NPM$	n[2*ndim +1*(ndim*P1)]	n[5*ndim +1*(ndim*P1) +1*((ndim+1)*P2)]	n[3*ndim +1*(ndim*P1)]
/	$NDIM \cdot NCAG \cdot NPM$	0	0	0
*	0	n[ndim]	n[ndim +1*P2)]	n[ndim + 2]
=	?	n[ndim]	n[ndim+1]	n[ndim]

Table 3.5 Number of operations to calculate the geom. centre and to construct the pairlist

ncag number of charge groups in a solute molecule

npm total number of solute molecules

n see eqn (3.8).

P1, P2 see table 3.4

The dynamic data for the pairlist construction are all coordinates as input, as output the pairlist pointer list and the pairlist itself. Thus the formula for the memory requirement reads as

$$M_{pairlist} = ndim \cdot nattot + CG_{solv} + NPC\,. \tag{3.9}$$

3.4.3 Best Case Speed-up's

With the results of the profiling section we can calculate the best-case speed-up in dependence on the function mapping. Functions to be calculated on a coprocessor have zero execution time, we neglect the communication and additional program overhead. For various simulation runs with different parameter sets (pairlist update rate, cutoff radius), we found the following results (table 3.6), where the first column means that only the solvent-solvent part of the nonbonded forces calculation is mapped on the coprocessor. If the pairlist construction is also performed on dedicated hardware, we have the results in the middle column. If the host is relieved of all nonbonded forces and the pairlist calculation we have the maximum speed-up rates in the third column.

run	solv-solv	s-s, pairlist	nonb, pairlist
5/1.4, thr2rect	**2,6**	**6,8**	**45**
5/0.8, thr2rect	1,4	9,1	18
10/1.4, thr2rect	3,4	6,5	41
10/0.8, thr2rect	1,7	6,5	13
5/1.4, thr2octa	**2,4**	**3,9**	**72**
5/0.8, thr2octa	1,5	6,3	23
10/1.4, thr2octa	2,8	3,6	70
10/0.8, thr2octa	1,8	4,5	18

Table 3.6 Max. speed-ups for thr2 with water, rectangular and octahedron boundaries

The number in the left-most column indicates the pairlist update rates (five or ten), the cutoff radius (unit nm, without twin-range), and the type of periodic boundary conditions (octahedron or rectangular), refer to table 3.1 for the system sizes. The benchmark without water is not part of this table because this kind of simulation is fast enough and would not be significantly accelerated with dedicated hardware.

3.5 Pairlist Algorithms

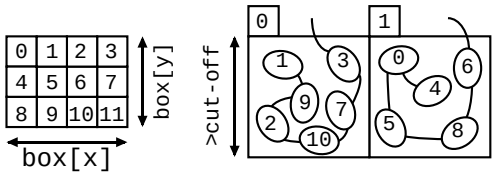
We implemented the classical Verlet linked list algorithm with a few changes to speed-up the pairlist calculation.

3.5.1 Cell Index Method

Under the Verlet algorithm an integration scheme as stated from Verlet is understood, the direct integration of Newton's second law of motion. Verlet was also the first who proposed the use of a pairlist. Verlet proposed a cutoff radius for the interaction and a small skin around the cutoff sphere to automatically update the pairlist if necessary. A brute force approach was used to calculate the pairlist by scanning all pairs. 1975 Quentrec and Brot proposed the cell index method where the simulation box is divided into a regular lattice of

cubic or rectangular cells with a side length of at least the cutoff radius. The cell structure may be built up using the linked list method, first proposed by Knuth 1973.

Since then big efforts were made to find more efficient algorithms for parallel computers or vector machines [10][11][13][14], in the field of sequential algorithms the progress was not overwhelming.




```
index 0 1 2 3 4 5 6 7 8 9 10
head  3 6
chain 4 -1 9 7 -1 0 8 10 5 1  2
```

Fig 3.5 Linked list

We decided to implement at least the combination of the cell index method and the linked list as stated from Hockney and Eastwood 1981 (fig. 3.5) [19]. Actually our final implementation has some modifications to improve performance for homogenous particle systems (liquids). In paragraph 3.5.2 we introduce another often used approach in sequential pairlist calculations, the grid cell method.

The Algorithm

First, we divide the box into cells such that the sides of the cells is greater than the cutoff radius. When calculating the pairlist, not all neighbours must be checked but only these in neighbouring cells. Then all particles are spatially sorted into their appropriate cell. Two arrays are created during the sorting process: The *head array* has one element for each cell containing the identification number of that particle (charge group). This number is used to access the second element of this cell in the *chain array,* which contains the number of the next charge group and so on. A “-1“ (minus one) indicates that there are no more elements in this cell.

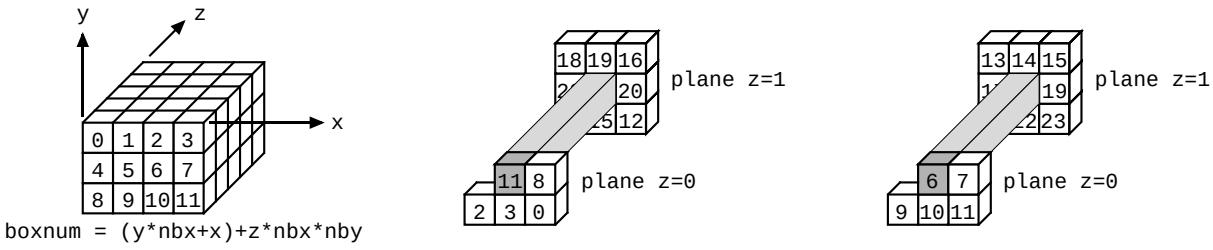


Fig 3.6 Pairlist numbering scheme

The sequential numbering of the charge groups (identification number) and the numbering of the cells as in fig. 3.6 are conventions and may not be altered

Because of eqn (3.5) not all neighbour boxes must be searched but only these indicated in fig. 3.6. Note how periodic boundaries are handled (neighbours for cell 11). After the sort into cells we need an algorithm to find the neighbouring cells for a given primary cell. This task is easy and fast for our cell numbering scheme.

For large cutoff spheres the number of cells decrease, and most of the investigated pairs do not fulfil the cutoff criteria. This leads to bad acceleration rate for large cutoffs compared with the brute force approach. To increase the efficiency of the algorithm under this circumstances we further divide the cells into sub-cells, called parts (fig. 3.7). As a consequence, the number of superfluous distance calculations is minimized. With other words, N_C (see next paragraph) is kept small for large cutoffs. The algorithm to get the neighbour cells is similar.

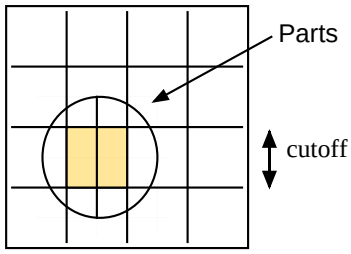


Fig 3.7 Parts

Implementation and results

We implemented the described algorithms in C and tested them in the Gromos environment with the water benchmark under rectangular boundaries. For a cutoff of 1.4nm there are 100 cells (the box length is approximately 7nm, leading to 4x5x5 cells). With the number of charge groups per cell N_C and the total number of charge groups N, the maximum speed-up is

$$A_{max} = \frac{N}{27 \cdot N_C}. \qquad (3.10)$$

Fig. 3.8 shows the measured speed-up related to the brute force algorithm without parts. The discrepancy between the maximum possible speed-up and the measurement can be explained with program and memory allocation overhead. In addition, we did not made every effort for a highly efficient implementation.

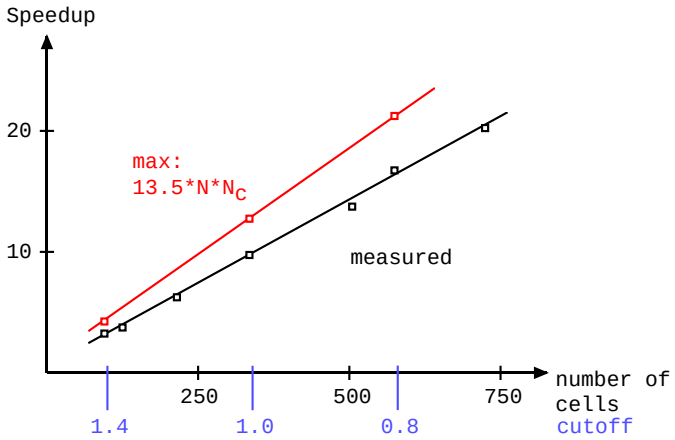


Fig 3.8 Speed-up with new pairlist algorithm

The implementation with parts allows the user to specify the part-factor, the number of parts per cutoff, as an integer. If the specified number is too big, e.g. a lot of empty cells occur, the algorithm automatically decrements the part-factor such that the number of cells is limited to the number of charge groups in the system. With this modification we achieve additional 20% for large cutoff (1.4nm in fig. 3.8), for small cutoffs no gain was determined.

3.5.2 Grid Search Algorithms

Similar to the extended linked list method only pairs with a cutoff smaller than a certain radius are stuck into the pairlist. A much finer grid is used to divide the simulation space into cells so that at most one particle is assigned to a cell. The cutoff sphere for a primary cell is approximated much better and the number of unsuccessful distance checks is minimized. The disadvantage are the many empty cells because the grid granularity is determined through the minimal distance of the particles which usually is much smaller than the average, even for liquids. That's why the grid is chosen coarser causing multiple occupancies. The surplus particles are assigned to the next empty neighbour cell. With this technique, the number of distance checks is kept to a minimum. Periodic boundary conditions are handled as usual, the simulation box is repeated at the edges. The algorithm sequence can be outlined as follows:

The first step is the sorting of particles into cells and the reassignment for double occupancies. For a given primary cell containing a particle the neighbour cells are determined. For that, a similar method as for the parts scheme may be used (numbering scheme), a second possibility is the use of eqn (3.11).

$$r_{min}^{d0} = \sqrt{(a_x \cdot \max(|d_x| - 1, 0))^2 + (a_y \cdot \max(|d_y| - 1, 0))^2 + (a_z \cdot \max(|d_z| - 1, 0))^2} \tag{3.11}$$

Considering a pair of cells with the distance (d_x,d_y,d_z). If the lattice constants are a_x,a_y,a_z, the minimal distance between two particles located inside these boxes is r_{min}^{d0}. To get the neighbours of a particle in a primary cell, all neighbouring cells inside the cutoff rectangle around the primary cell are checked for $r_{min}^{d0} < r_{cut-off}$. If the check is true, the distance between the corresponding particles must be calculated to determine if the particular neighbour goes into the pairlist or not.

The remaining tasks do not vary from the pairlist construction mentioned above. An implementation example and performance data can be obtained in [11]. We did not implement this algorithm, because the gain to the parts algorithm is quite small for big cutoffs, and the implementation effort would be rather high. For smaller cutoffs all pairlist algorithms except the brute force approach are enough efficient. For more information about pairlist algorithms refer to: [10], a comparison of different algorithms); [11], a description of a grid search algorithm); [21] and [22].

4 Dedicated Hardware Approaches

4.1 Overview on Existing Third-Party Solutions

Four hardware solutions for the molecular dynamics problem were recently developed: MD-GRAPE, GRAPE-4, Gromacs and MD-Engine. They are described in more detail in the following sections. Other older hardware projects are not described in this report for information obout these projects refer to [1] and [3].

4.1.1 GRAPE

GRAPE was developed by the Department of Earth Science and Astronomy together with the Department of Information Science and Graphics of the College of Arts and Sciences at the University of Tokyo (refer to [4], [5], [6] and [8]).

GRAPE (for GRavity PipE) is a coprocessor attached to a general purpose computer and is specialized to calculate the interactions between particles of a N-body system. GRAPE has pipelines specialized for force calculations, which is the most expensive part of a N-body gravity simulation. All other computations, such as time integration of orbits etc. are performed on the host computer to which GRAPE is connected. In the simplest case, the host computer sends positions and masses of all particles to GRAPE. Then GRAPE calculates the forces between particles and sends them back to the host computer (Fig. 4.1).

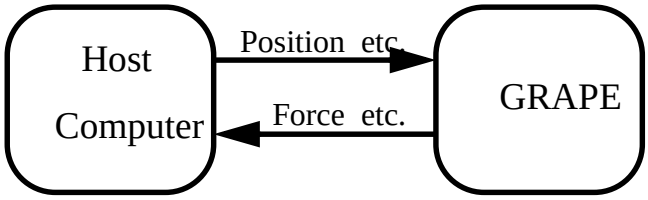


Fig 4.1 Basic structure of the GRAPE system

N-body simulations are usually performed with periodic boundary conditions to express global homogeneity without margin conditions. The most widely used algorithm for the force calculation under periodic boundary condition is the particle-mesh (PM) method, which cannot be accelerated on GRAPE. Thus, for GRAPE, a simulation method has been developed which requires the calculation of particle-particle (PP) forces.

However the PP force is not a pure $1/r^2$ force. Therefore there are two ways to implement the PP force calculation on GRAPE. One is to mimic the PP force by a linear combination of $1/r^2$ forces based on several approximations, which results in rather large loss of accuracy and performance. A better solution is to change the hardware such that it can handle force laws other than $1/r^2$. With this, GRAPE-2A has been developed out, but the peak performance is rather low (180 Mflops) since GRAPE-2A is made of commercial floating point chips [5]. To improve the system, MD-GRAPE was developed, which is discussed in the next section.

An additional version of MD-GRAPE has been built later by the same departments at the University of Tokyo and is called GRAPE-4. This architecture is discussed in paragraph 4.1.3.

4.1.2 MD-GRAPE

Overall Hardware Architecture

The MD-GRAPE system consists four MD chips, a particle index unit, a memory unit and an interface unit and is assembled on a single board (Fig 4.2).

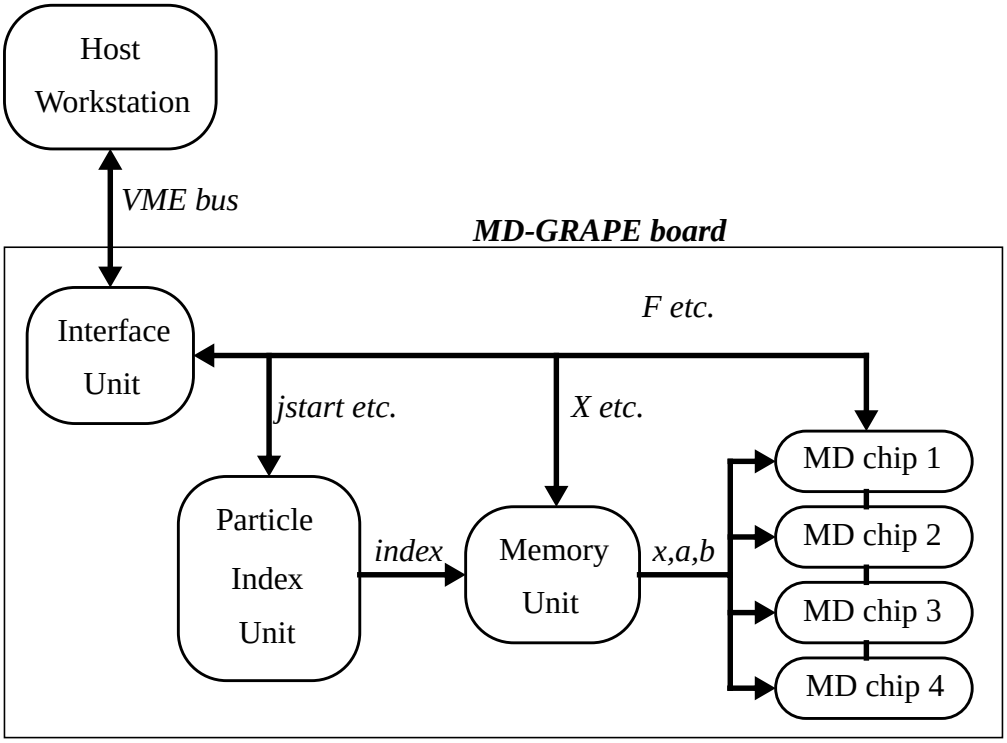


Fig 4.2 Hardware architecture of the MD-GRAPE system

Algorithms

Three methods are used to calculate the gravitational force under the periodic boundary condition on MD-GRAPE.

The $O(N^2)$ direct summation algorithm with free boundary conditions is the simplest force calculation with free boundary conditions. Positions of particles are sent to MD-GRAPE, which calculates the forces in force mode. The forces are sent back to the host, which calculates the new positions and sends them to MD-GRAPE again and so on. In this calculation position vectors r_i are distributed to MD chips whereas r_j are sent to all chips.

The next two methods divide the force into real space (short-range part) and wavenumber space (long-range part). Whereas the short-range part is calculated directly, the long-range part is calculated with help of the Fourier transform.

The particle-particle/particle-mesh (P^3M) method is a refinement of the PM method. It utilizes the fast Fourier transform (FFT) for calculating the long-range part. In the P^3M method, MD-GRAPE calculates the PP force, while PM force is calculated on host computer. In order to use FFT, the P^3M method assigns masses to mesh points. Therefore, the forces calculated by the P^3M method include errors due to the mesh assignment. The positions are updated on the host computer by this method.

The Ewald method calculates the gravitational force under the periodic boundary condition with high accuracy and was developed to obtain the Madelung constant of an ionic crystal. In contrast to the P^3M method the discrete Fourier transform (DFT) is used to calculate the long-range part. So the correct force under the periodic boundary is obtained. Both forces, the one of the real space and the one of the wavenumber space can be calculated by MD-GRAPE, whereas the cell-index method is used for real space forces and DFT/IDFT is used for wavenumber space forces.

An arbitrary central force is calculated with the following equations:

$$f_i = \sum_j^N f_{ij} = \sum_j^N a_j g(b_j r_s^2) r_{ij} \tag{4.1}$$

$$r_s^2 = r_{ij}^2 + \varepsilon^2 \tag{4.2}$$

where ε is a softening parameter, a_j and b_j are coefficients, and $g(\zeta)$ expresses an arbitrary function. Index i is used for the particle at which the force is calculated and index j for particles which exert the forces on particle i.

The potential is given by:

$$f_i = \sum_j^N f_{ij} = \sum_j^N a_j g(b_j r_s^2) \tag{4.3}$$

In DFT/IDFT mode, the sum is calculated by:

$$f_i = \sum_j^N f_{ij} = \sum_j^N a_j g(k_i \bullet r_j) \tag{4.4}$$

Detailed Hardware Architecture

Each MD chip integrates a GRAPE-2A pipeline. This “virtual multiple pipeline” architecture reduces the bandwidth necessary for the data transfer during force calculation. This hardware pipelines act as if multiple virtual pipelines were operating at a slower speed. One hardware pipeline has six “virtual” pipelines and calculates forces on six particles f_i,..., f_{i+5} simultaneously. Therefore the partial forces f_{ij} are evaluated in the following order:

$$f_{11}, f_{21}, f_{31}, f_{41}, f_{51}, f_{61} \;\big|\; f_{12}, f_{22}, f_{32}, f_{42}, f_{52}, f_{62} \;\big|\; ..... \tag{4.5}$$

One interaction is evaluated in each clock cycle. The MD chip calculates the forces on six different positions of the system using the same position vector r_j and coefficient vectors a_j and b_j but a different r_i. Therefore one position vector and two coefficients are supplied to the MD chip during six clock cycles (Fig. 4.3).

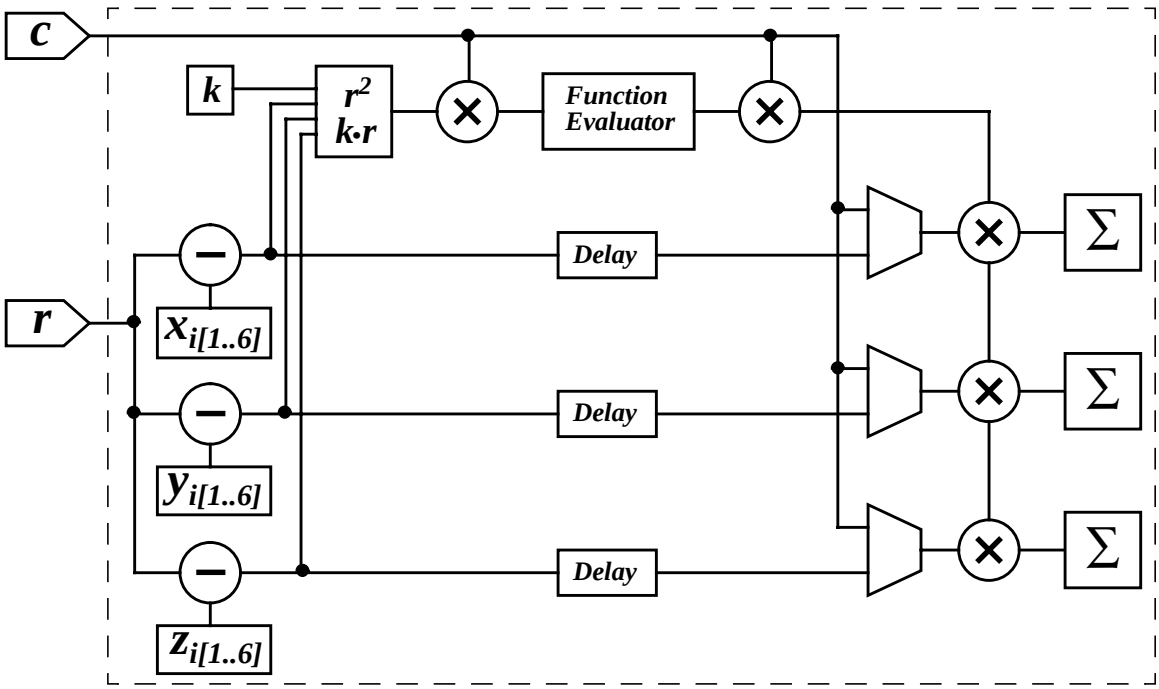


Fig 4.3 Block diagram of the MD chip

The MD chip can be used in three modes: in force mode, in potential mode and in DFT/ IDFT mode. In force mode eqn (4.1) and eqn (4.2) are directly calculated by the MD-chip. In potential mode eqn (4.3) is calculated and in DFT/IDFT mode eqn (4.4) is calculated directly.

The coordinates *x, y, z,* of the position vectors r_i and r_j are given through the ***r**-port* and coefficients a_j and b_j are given through the ***c**-port* of the chip. At the beginning of a sum calculation the position vector and the constant *k* in DFT/IDFT mode of particle *i* are given through the ports. During calculating position vector and coefficients of particles *j* are given through the ports.

The MD chip has eight multipliers, nine adders and one function evaluator. The function evaluator contains the arbitrary function g(ζ). Positions are expressed in 40 bit fixed point format. The internal calculations are performed in 32 bit floating-point format, and the accumulation of the force is done in 80 bit fixed-point format. The function evaluator is a look-up table (LUT) with 1024 values.

The particle index unit supplies particle indices to the memory unit. It is optimized for cell-index (paragraph 3.5.1) mode, which is used to calculate the short-range force.

The simulation cube is divided into M^3 cells, where M is the number of cells along one dimension. M is given by the largest integer less or equal to L/r_{cut}, where r_{cut} is the cutoff length of the force and L is the length of the periodic boundary box. In order to calculate the forces between particles in a cell, only contributions from particles of the 27 neighbouring cells (including the cell in question) have to be calculated. Therefore the computation cost is reduced by a factor of $M^3/27$ (Fig. 4.4).

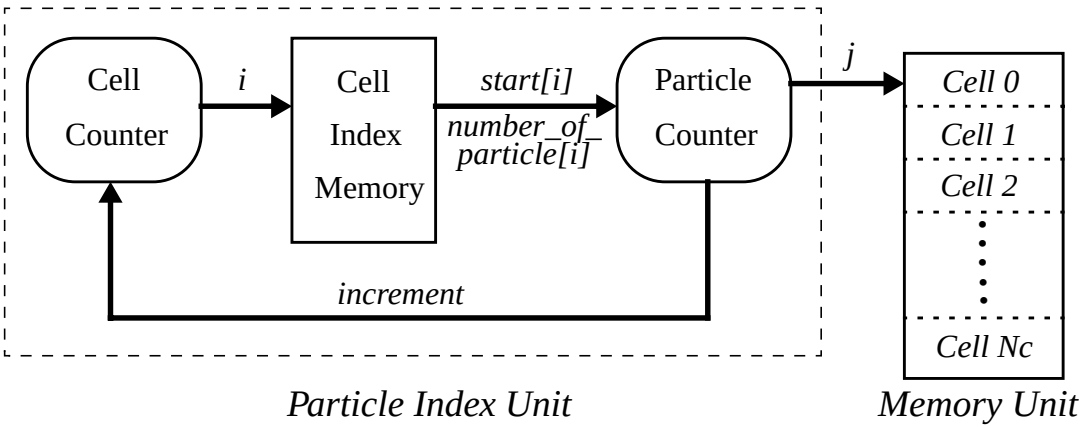


Fig 4.4 Structure of the particle index unit

The particle index unit contains two counters: a cell counter, which is a 14 bit counter and a particle counter, which is a 17 bit counter. The particles in a cell are stored in consecutive locations in the memory unit. The cell index memory, which is a 1 Mbit (16K x 32bit) SRAM module, supplies addresses and number of particles per cell, with which a cell interacts. Initially, cell number *i* is 0 and *j* is set to *start[i]*, which points to the start address of the present cell. Then the data of the memory unit on which *j* points are sent to the MD chips and *j* is incremented. This steps are repeated until *j* reaches the address *start[i]+number_of_particle[i]*. At this point *i* is incremented by one and *j* is set to *start[i]*, which points to the first address of the next cell. This procedure is repeated until *i* reaches the maximal number of cells. In this way correct coordinates of particles are automatically sent to MD chips.

The memory unit supplies the data of positions r_j and coefficients a_j and b_j to the MD chip according to the particle indices supplied by the particle index unit. The data r_j, a_j and b_j are shared by all the MD chips. The memory unit is composed of nine 1 Mbit (128K x 8) SRAMs. Five are for the position vectors and four are for the coefficients. It can hold up 43,690 particles in three dimensions.

The interface unit is a VME-bus ‘slave’ interface, which transmits and receives data according to the request of the host computer. The interface interacts with the host by handshake, writes the received data, such as r_j, to the location specified by the address and supplies data, as the calculated force, with a read cycle.

The MD-GRAPE has a neighbour list unit. The MD chip outputs the “neighbour flag’ if the distance between particles r_{ij}, is less than the neighbour radius h_i, which is stored in a register on the MD chip. This flag is used to construct the neighbour list, respectively the pairlist of particle *i*.

Performance

On the MD-GRAPE board a N-body simulation with the Ewald method takes $600(N/10^6)^{3/2}$ seconds per step; the P^3M method takes $240(N/10^6)$ seconds per step.

Each MD chip calculates 3.5 x 107 interactions at a clock frequency of 35 MHz. If we count the square root and division operations as 13 floating-point operations, the calculation of

one gravitational force corresponds to 30 floating-point operations. An MD-GRAPE board has a peak performance of 4.2 Gflops.

Further details of MD-GRAPE can be found in [7].

4.1.3 GRAPE-4

Overall Hardware Architecture

An additional version of MD-GRAPE has been built by the same departments at the University of Tokyo and is called GRAPE-4.

GRAPE-4 is a massively parallel computer which consists of 1672 processor chips. Each processor chip integrates about 15 floating point arithmetic units and one function evaluator.

The basic structure of GRAPE-4 is the same as of MD-GRAPE (Fig. 4.1) and is used as a backend processor, to calculate the forces. The rest of the computation, such as the actual orbit integration is done on the host computer. In the simplest case, the host computer sends positions and masses of all particles to GRAPE, GRAPE calculates the forces between particles and sends them back to the host computer.

The GRAPE-4 system consists of a host computer and four GRAPE clusters. One cluster consists of a host-interface board and nine processor boards, thus the total number of processor boards is 36 (Fig. 4.5).

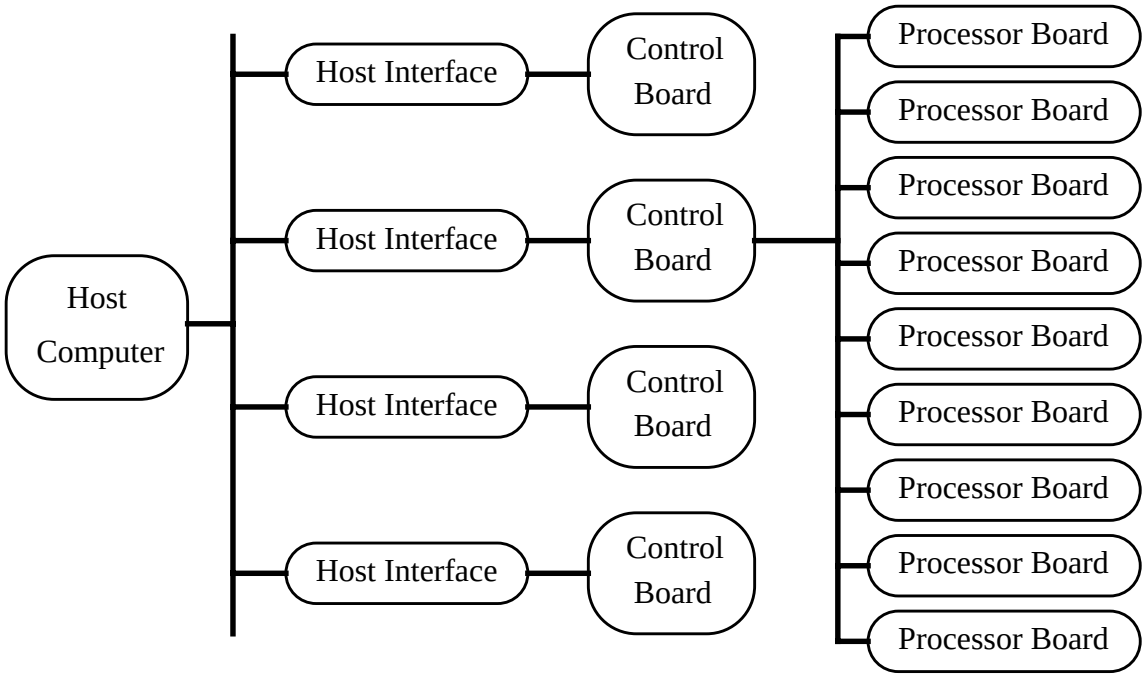


Fig 4.5 Structure of the GRAPE-4 system

Algorithms

The Aarseth scheme has been widely used in the community of Astrophysics. It is a linear-multistep scheme, in which the change in the position and velocity is calculated from the acceleration calculated at several previous timesteps. The implementation of the Aarseth scheme is rather complicated, because it is a linear multistep scheme. When the time inte-

gration is started, the accelerations at previous timesteps are not available. Therefore a special procedure to start up the integration is required.

The Hermite scheme is much simpler than the Aarseth scheme and yet offers a similar accuracy for the same calculation cost. It uses the Hermite interpolation formula to construct the predictor and the corrector. The predictor uses the position, velocity, acceleration and its derivative at time *t*. The acceleration and its time derivative are calculated from the position and velocity. The corrector requires information of the present and old timesteps only. Therefore no information concerning the previous timesteps is necessary. The Hermite scheme is a one-step, self starting scheme. It uses only the first time derivative of the acceleration explicitly calculated in order to construct the predictor and the corrector.

The predictor is expressed as:

$$x_p = \frac{\Delta t^3}{6} + \frac{\Delta t^2}{2} + \Delta t v_0 + x_0 \tag{4.6}$$

$$v_p = \frac{\Delta t^2}{2}\dot{a}_0 + \Delta t a_0 + v_0 \tag{4.7}$$

where x_p and v_p are the predicted positions and velocity, x_0, v_0, a_0 and $\dot{a}_0$ are the position, velocity, acceleration and its time derivative at time t_0, and Δt is the timestep.

The acceleration a and its time derivative $\dot{a}$ are calculated as:

$$a_i = \sum_j G m_j \frac{r_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}} \tag{4.8}$$

$$a_i = \sum_j G m_j \left(\frac{v_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}} - \frac{3(v_{ij} \cdot r_{ij}) r_{ij}}{(r_{ij}^2 + \varepsilon^2)^{5/2}} \right) \tag{4.9}$$

where

$$r_{ij} = x_j - x_i \tag{4.10}$$

$$v_{ij} = v_j - v_i \tag{4.11}$$

The corrector is given by the following formulas:

$$x_C = x_0 + \frac{\Delta t}{2}(v_C + v_0) - \frac{\Delta t^2}{2}(a_1 - a_0) \tag{4.12}$$

$$v_C = v_0 + \frac{\Delta t}{2}(a_1 + a_0) - \frac{\Delta t^2}{2}(\dot{a}_1 - \dot{a}_0) \tag{4.13}$$

Detailed Hardware Architecture

The host interface board extends the I/O bus of the host. It converts the data transfer protocol of the host I/O bus to the protocol used on the links between the host interface boards and the control boards. The protocol on the link is designed such that it is independent on the protocol of the host I/O bus. In this way, GRAPE-4 can connected to different host computers (Fig. 4.6).

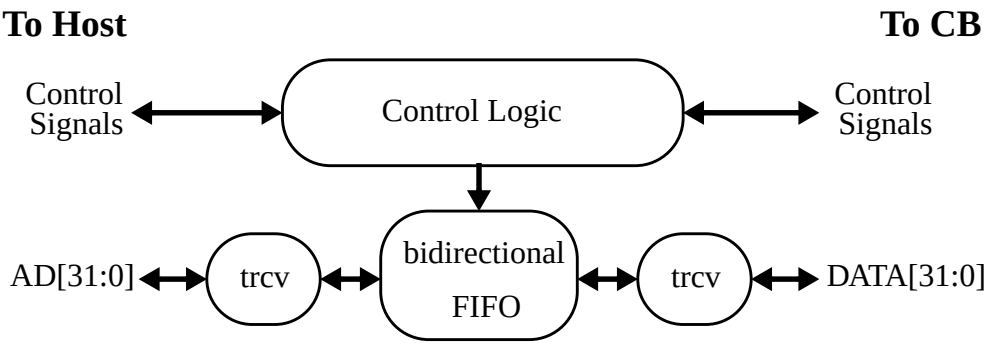


Fig 4.6 Structure of the host interface

The data transceivers (trcv) exchange data with the host, the control board, the FIFO, which is 2048 x 32-bit words (8 Kbytes), and the control logic. The data transfer rate between the host interface an the control board is slower than that between host and host interfaces. Using the FIFO buffer, the host can send data at peak transfer speed. At the other side, the FIFO buffer allows the control board to transfer data without checking the status of the host bus. Therefore it works independent of the host bus.

There are two different ways to use multiple GRAPE-4 boards. One is to let all chips calculate the force on different particles from the same set of particles. In this case, the content of the memory of all processor boards would be identical. The other way is to let each processor board to calculate the force on the same set of particles, but from different set of particles. In this case, each processor board calculates the partial forces which need to be added together with results obtained from other boards.

The first approach is not practical in the case of GRAPE-4 with more than 1000 pipelines, so the latter approach was implemented. Here the problem is that the amount of communication is proportional to the number of processor boards, if the single results are sent back to host, which adds them together. To solve this problem the control board was designed, which can add results calculated on processor boards to reduce bandwidth of communication with host computer.

The control board has two main functions. The first is to distribute the data received from the host computer to the processor boards. The second is to sum up the forces and potentials calculated on processor boards and then to transfer them to the host computer (Fig. 4.7).

The internal structure of the cluster is not visible to the application program. To the host computer, a cluster looks like a board with a single memory unit and multiple pipeline chips. In order to distribute the calculation over different clusters, the same is used as different processor boards in one cluster would be used. If we have 4 clusters, the host sends N/4 particles to each cluster, where N is the total number of particles. In this case each processor board takes care of the contributions from N/36 particles. The control boards adds the par-

tial forces calculated on processor boards and the host computer adds finally the partial forces calculated on clusters. The control boards consists of the control logic unit, three accumulator/buffer units and several transceivers and buffers.

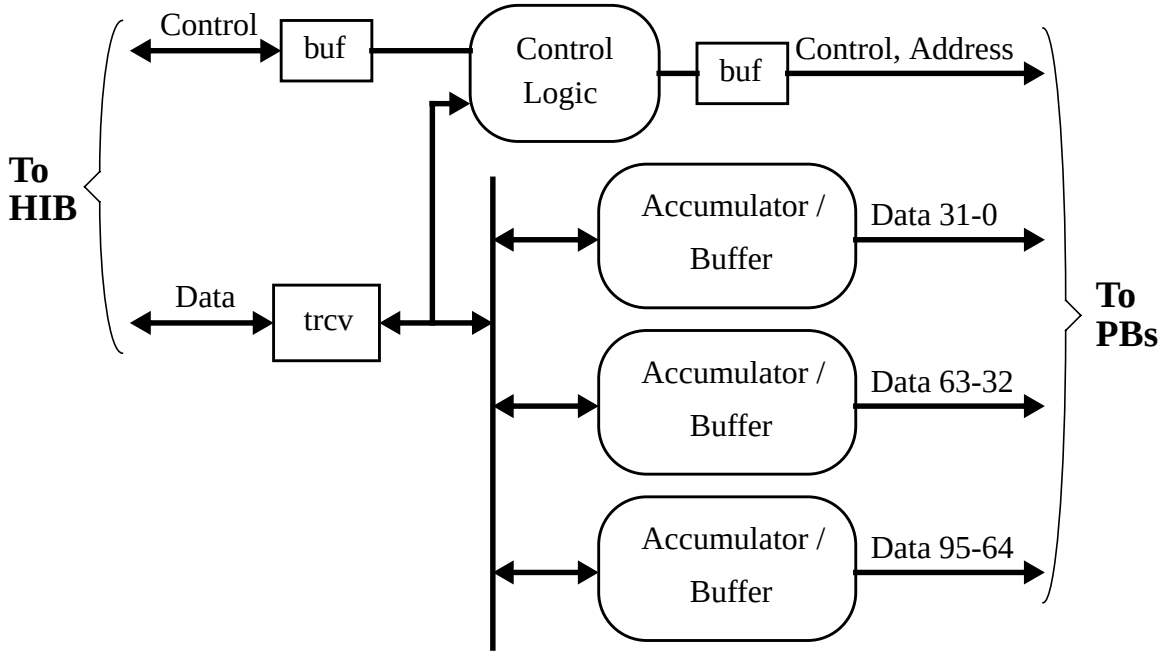


Fig 4.7 Structure of the control board

Control boards and processor boards are connected with a 96-bit data width backplane bus, which is called the HARP-bus. A synchronous, pipelined protocol with fixed latency is used on this bus. All necessary signals to control the HARP-bus are generated by the control logic unit. The 96-bit data bus is divided into three 32-bit subbuses, each of them is connected to an accumulator/buffer unit. The accumulator/buffer unit contains a 64-bit floating-point ALU, which accumulates the result calculated on processor boards.

Each processor board (fig. 4.9) of GRAPE-4 consists of 48 HARP chips. A block diagram of a HARP chip is shown in fig. 4.8. All 48 chips form multiple pipelines which share one particle memory unit. These pipelines calculate the forces between different particles from the same set of particles.

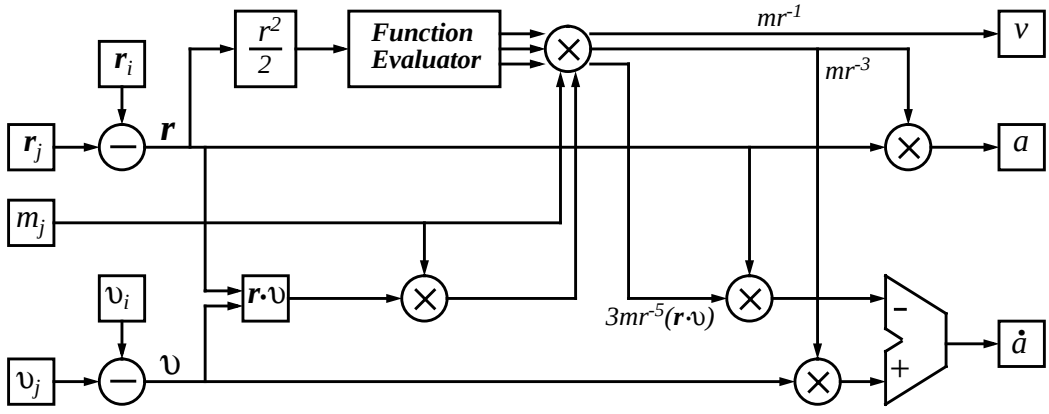


Fig 4.8 Structure of the HARP chip

The particle data memory stores the data of the particles which exert the force (Fig. 4.9).16 HARP chips share the same data bus such that three 32 Bit data busses are connected with the control board. The HARP chips are custom LSI chips which calculate the gravitational force and its first time derivatives for particles. This fully-pipelined hardware implementation of eqn (4.8) and eqn (4.9) is shown in fig. 4.8. The *x* , *y* and *z* components of all vector quantities are processed sequential in order to reduce the gate count.

Subtraction of the position vectors and accumulation of the calculated accelerations are performed in 64-bit floating-point format. Other calculations to obtain the acceleration is performed in 32-bit format. Subtraction of the velocity vectors and accumulation of the time derivatives are performed in 38-bit format. Other calculations to obtain time derivative are done in 29-bit format, where the length of the mantissa is 20 bits.

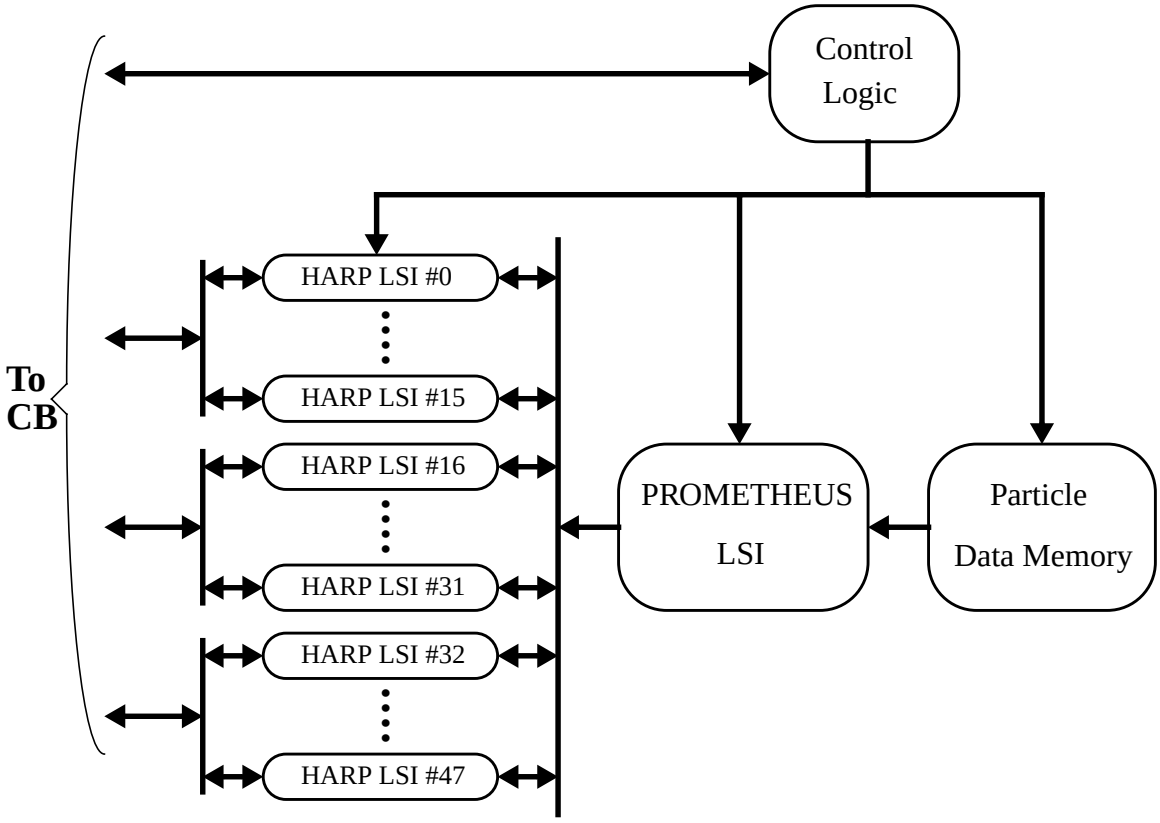


Fig 4.9 Structure of the processor board

The hardware is fully pipelined and therefore it takes three clock periods to calculate one interaction. Again each chip calculates the forces between two particles, using the “virtual multiple pipeline”. The clock period of the pipeline is two times that of the system clock and calculates the forces between two different particles at alternate clock cycles. The chip has two separate set of registers and the two virtual pipelines operate independently. From the outside, one force calculation chip looks as if it has two pipelines. The advantage of this architecture is that we can increase the performance of the pipeline chip without increasing the system clock cycle.

The PROMETHEUS chip (fig. 4.10) is another custom LSI-circuit and is used to predict the position and velocity of particles at a specified time. The PROMETHEUS chip handles x, y and z components sequential in the same way as the HARP chip does. The predicted posi-

tion and velocity are calculated based on eqn (4.6) and eqn (4.7). The calculation of Δt and the addition of x_j and the higher order term are performed in 64-bit format. All other calculations are performed in 32-bit format. The clock of this chip is the same as the system clock of the processor board.

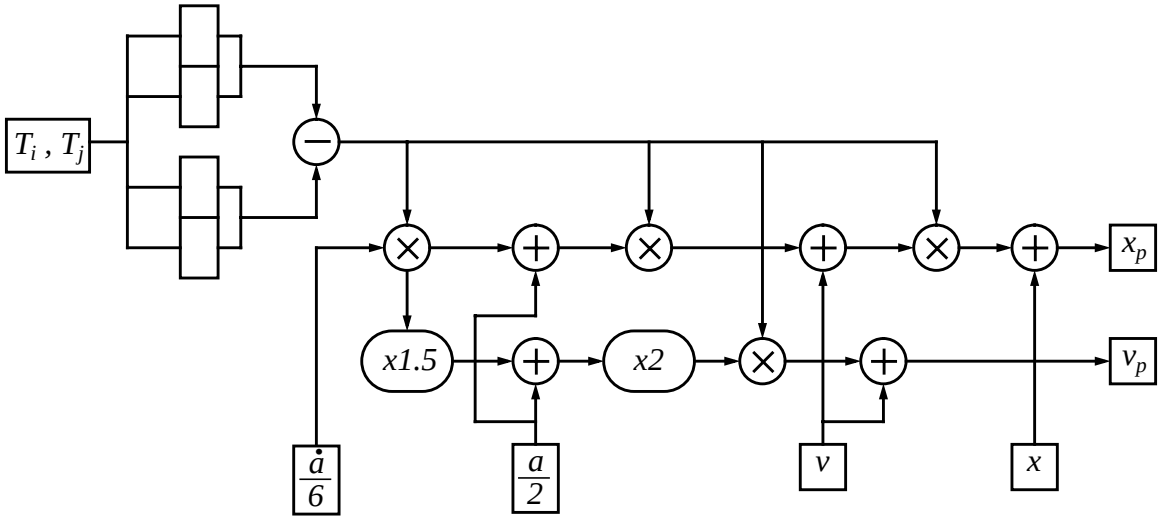


Fig 4.10 Structure of the PROMETHEUS chip

The communication between the host interface board and the host computer relies on DMA transfers. An interface library for UNIX has been developed, to allow the application to invoke DMA transfer to/from its memory space directly.

Performance

GRAPE-4 integrates about 35,000 floating-point arithmetic units. The peak performance of a chip is 640 Mflops and that of the total machine is 1.08 Tflops for the total cost of less than 2 million dollars.

For the next generation of GRAPE-4, a machine with about 200 Tflops is planned, which consists of 1,400 chips at a running clock frequency of 150 MHz. The number of transistors per chip will be 50 times larger than that of the HARP chip. Refer to [6] for more information about GRAPE-4.

4.1.4 GROMACS

The GROMACS hardware and system software [9] was designed at the University of Groningen by the MD Group, in a joint project with Chess Engineering in Harlem, Netherlands.

Overall Hardware Architecture

The architecture consists of a ring of 32 processor nodes. Each processor node consists of a 40 MHz Intel i860 processor, 8 Mbyte RAM, two 8 bit wide parallel ports to connect one node to the ring, four transputer links and an 8 bit PC-bus connector.

Ring communication with a bandwidth of 3 Mbytes/sec is used for the actual simulation. The PC-bus is used to load programs and initial data and to store results, the transputer links are not used (Fig. 4.11).

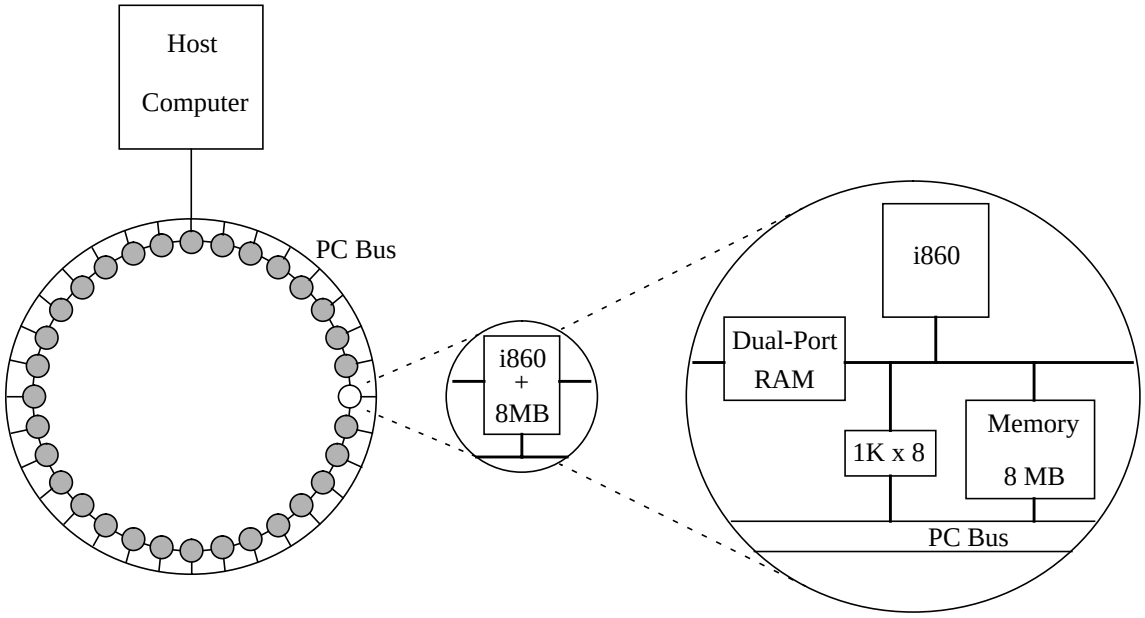


Fig 4.11 GROMACS system

Algorithm

GROMACS uses a grid search algorithm for generating the pairlist. Before the neighbour list is generated, a grid is constructed in the box. For every grid cell it is determined, which particles it contains. Neighbours of a particle are then searched by inspecting neighbouring boxes. Experience shows that a grid size $L=1/2R_{cutoff}$ results in optimal neighbour search. A finer grid gives a higher in range ratio but more visits to empty grid cells, leading to a lower neighbour searching speed.

Because the set of nonbonded forces changes every few time steps, the position of every particle has to be distributed over half of the ring. It is impossible to allocate particles to processors in such a way that communication remains minimal.

Distribution of particles over half of the ring is insufficient for three and four particle bonded force calculations because in this way not every position triple or quadruple is present on at least one processor. During a simulation the set of bonded force calculations does not change and the communication characteristics remain the same for a given allocation. So an allocation generated during a preprocessing phase remains valid during the whole simulation. It is such that particles in triplet and quadruple interaction get close numbers, so will be allocated on close processors.

Software

The GROMACS software consists of preprocessing software, which runs on the host and MD simulation software, which runs on the ring. The functionality includes Lennard-Jones, Coulomb and harmonic potentials, many types of bond-angle and dihedral interactions, different types of neighbour searching, notions like charge groups, exclusions, position and distance restraints, coupling to temperature and pressure baths and free energy calculation. During the actual simulation process an MD simulation can be monitored, interrupted, modified, resumed and stopped.

The software is designed to work on a ring of any size, even on a single i860 board. Scaling from 1 to 8 processors (tested on 2000 water molecules) gives negligible scaling overhead, scaling from 8 to 32 processors yields only an efficiency of 75%. Communication overhead is in the region of 10 to 15% for a ring of 32 processors. Based on these numbers one may expect 20 to 30% overhead for 64 processors and 40 to 60% for a ring of 128 processors.

Performance

GROMACS does (3.5 to 7) x 106 nonbonded force calculations per second on a typical problem including overhead of neighbour searching and bonded force calculations. It is 5-6 times faster than a CRAY and 3-4 times faster then the NEC SX-3 vector supercomputer. For more information refer to [9].

4.1.5 MD-Engine

Introduction

MD-Engine is a special purpose parallel machine for molecular dynamic calculations made by Fuji Xerox. In order to simplify the hardware, it was developed to calculate only non-bonded forces.

Overall Hardware Architecture

An MD-Engine system consists of a host computer and up to 4 MD-Engine system boxes. The host computer sends coordinates of particles or reciprocal lattice vectors to the MD Engine, then the MD-Engine calculates and sends back forces, the virial or a coefficient of reciprocal lattice vector (Fig. 4.12).

An MD-Engine system chassis may house up to 20 MD-Engine cards. On each card there are 4 special purpose processors called MODEL-chip (MOlecular Dynamics Processing ELement). The MODEL chip is responsible for all the required arithmetic operations to calculate the nonbonded forces.

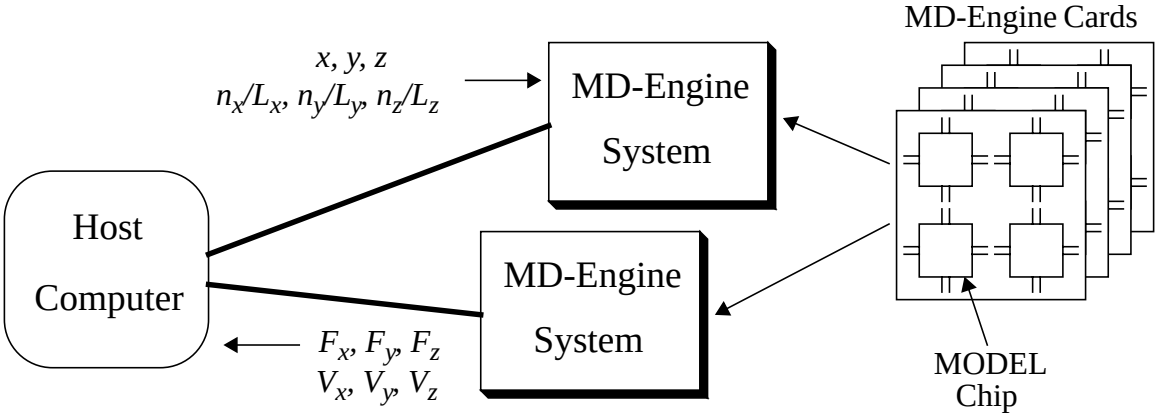


Fig 4.12 MD-Engine system and cards

Algorithm

To calculate the nonbonded forces the Van der Waals force and the Coulomb force are needed. The Van der Waals force is calculated by:

$$f_{Lij}(t) = E_{ij}\left\{2\left(\frac{r_{0ij}}{r_{ij}(t)}\right)^{14} - \left(\frac{r_{0ij}}{r_{ij}(t)}\right)^{8}\right\}\bar{r}_{ij}(t) \tag{4.14}$$

where $\bar{r}_{ij}(t)$ is a relative position vector between the *i-th* particle and the *j-th* particle, $r_{ij}(t)$ is the absolute value of $\bar{r}_{ij}(t)$, E_{ij} is a parameter of energy and r_{0ij} is a scaling factor of distance.

The Coulombic interaction is written as:

$$f_{Cij}(t) = q_i\left(\frac{q_j}{r_{ij}(t)^3}\right)\bar{r}_{ij}(t) \tag{4.15}$$

where q_i and q_j are the electric charges for each particle. When calculating Coulomb forces in free space, three virial elements are summed up to Coulombic potential.

The Ewald method is a way to calculate Coulomb force precisely, when electrically charged particles exist periodically. A rectangular box is modelled on the computer. The side lengths of this box are defined as L_x, L_y, L_z.

In eqn (4.16), eqn (4.18) and eqn (4.18) the *x* element of Coulomb force exerted on the *i-th* particle are expressed:

$$f_{ix1} = q_i\sum_j q_j\sum_{n'}\left\{\frac{2a}{\sqrt{\pi}}\exp(-(a\bar{r}_{ij,\bar{n}'})^2) + \mathrm{erfc}(a\bar{r}_{ij,\bar{n}'})\right\}\frac{x_{ij,\bar{n}'}}{\bar{r}_{ij,\bar{n}'}^3} \tag{4.16}$$

$$f_{ix2} = \frac{2q_i}{L_x^2}\sum_n \frac{\exp\left(-\pi^2|\bar{n}|/a^2L_x^2\right)}{|\bar{n}|^2}n_x\cdot\left\{\left(\sum_j q_j\cos 2\pi\bar{h}\bar{r}_j\right)\sin 2\pi\bar{h}\bar{r}_j - \left(\sum_j q_j\sin 2\pi\bar{h}\bar{r}_j\right)\cos 2\pi\bar{h}\bar{r}_j\right\} \tag{4.17}$$

$$f_{ix} = f_{ix1} + f_{ix2} \tag{4.18}$$

An integer vector ***n*** is defined as $\bar{n}=(n_x, n_y, n_z)$, where n_x, n_y, n_z are integer numbers. The position vector of the *i-th* particle's image can be written as $\bar{r}_{i,\bar{n}}=(n_xL_x+x_i, n_yL_y+y_i, n_zL_z+z_i)$. A reciprocal lattice vector $\bar{h}$ is defined as $\bar{h}=(n_x/L_x, n_y/L_y, n_z/L_z)$. α is an arbitrary positive number and erfc(x) is the complement error function defined as:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}}\int_0^x \exp(-t^2)dt \tag{4.19}$$

Detailed Hardware Architecture

MD-Engine is a single-bus multi-port local memory multiprocessor system. The SBus and VME bus interface card translates an SBus access request to a VME bus access request. All MODEL chips are connected to the VME bus in parallel (Fig. 4.13).

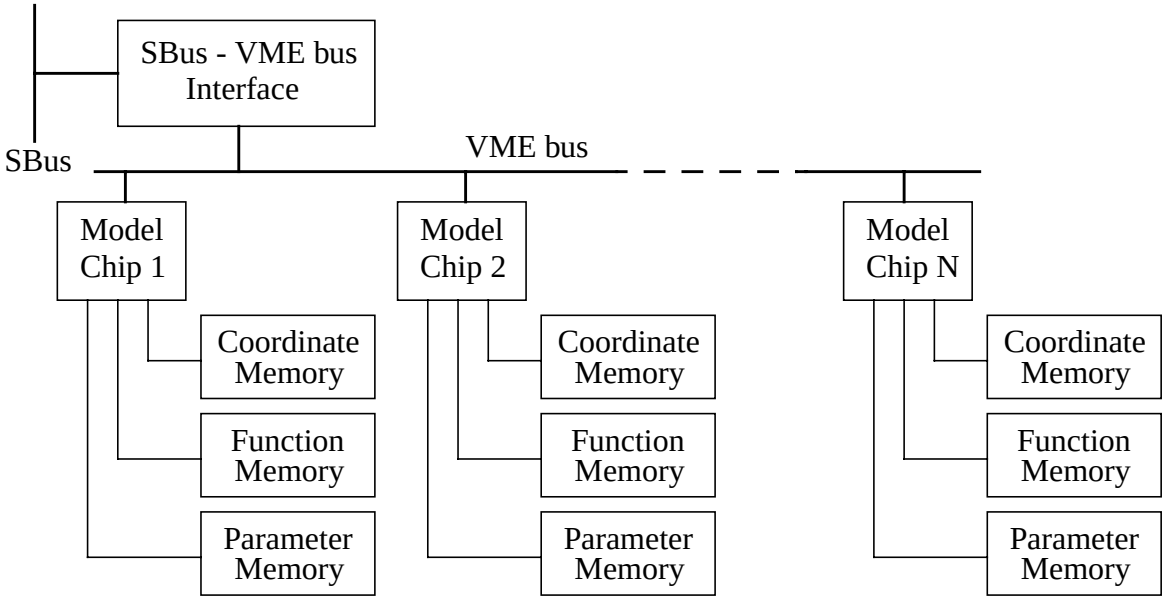


Fig 4.13 MD-Engine multiprocessor system

Three types of local memories are connected to the individual memory ports of the MODEL chips. The coordinate memory stores coordinates, electrical charges and index numbers of species of particles. The function memory stores the 4 sets of three coefficients for quadratic interpolation. The parameter memory stores 2 or 8 sets of force field parameters. These local memories are mapped to the internal register addresses of the MODEL chip.

The host computer can simply access internal registers of the MODEL chip and the local memories as SBus memory mapped devices. While executing an MD simulation, often the same data are written to the registers of the MODEL chips. The host computer can write the MODEL chip registers at once using a global chip address. Before starting force calculation, the host computer broadcasts coordinates of all the particles, force field parameters and coefficients of interpolation to the appropriate local memories of the MODEL chips. Only the coordinate memories have to be updated at each time step. The MODEL chip carries out all particles *j* to obtain the force acting on the *i*-th particles. The force calculations of the *i* particles are equally distributed on all MODEL chips. The coordinate memories of every MODEL chip contains the coordinates of all particles. VME bus is not used during the force calculation. Furthermore, a MODEL chip is able to access its three local memories simultaneously, which allows for parallel operations.

The goal of the MODEL chip was not only to calculate the values with enough accuracy, it should also compute very fast, in addition minimal hardware design and costs was desirable.

With the MODEL chip Van der Waals and Coulomb forces (eqn (4.15) and eqn (4.14)) can be calculated directly. The Coulomb force can also be calculated with the Ewald method (eqn (4.16), eqn (4.18), eqn (4.18) and eqn (4.19)). In addition the potential energy of the system can be evaluated to ensure that the simulation is running correctly (Fig. 4.14).

The format of the position vector is defined as 40-bit floating-point format with 31 bits mantissa. 64-bit double floating point format with 52 bits wide mantissa is used to accumulate pairwise forces and the virial. There are four 40-bit floating-point adders, three 40-bit floating-point multipliers and two 64-bit floating-point adders in a MODEL chip.

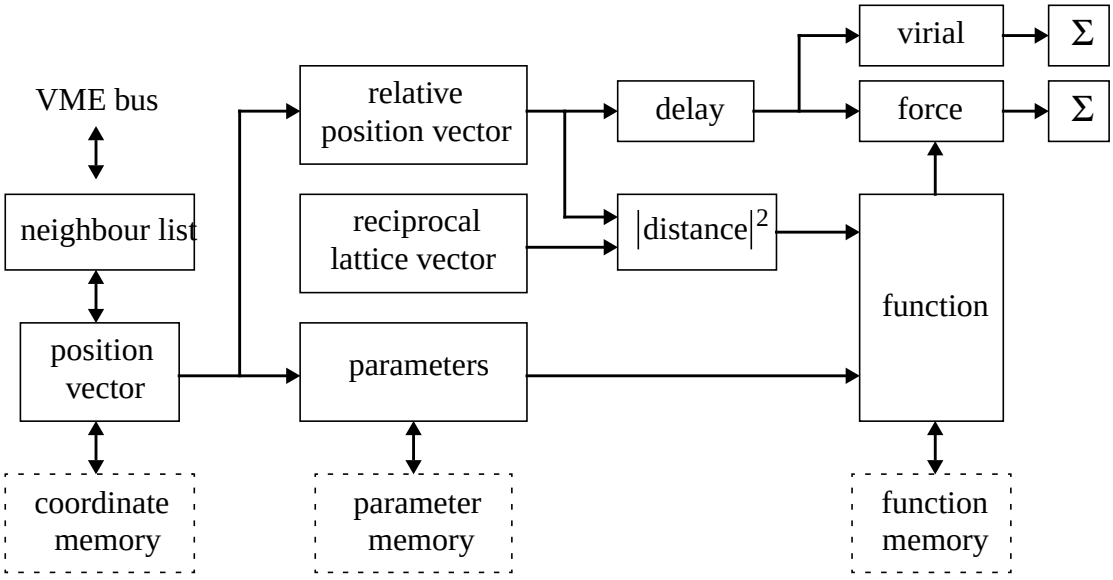


Fig 4.14 Functional block diagram of MODEL chip

When calculating Coulombic force all position vectors and electrical charges are fetched from the coordinate memory. Coulombic force and virial are calculated simultaneously. The reciprocal lattice vector and parameter memory are not used. When calculating Van der Waals force, a neighbour list scheme can be used, which may be generated by the MODEL itself while calculating Coulombic force or by the host computer. The reciprocal lattice vector is not used in this mode. In order to calculate the Ewald sum, the MODEL chip is used in five steps and the parameter memory and neighbour list are not used.

There are three library classes delivered with the MD-Engine. The low level and interface libraries and the application program. The functions in the low level library control the hardware of the MD-Engine. SBus devices are mapped into user virtual addresses. The function supports broadcasting data to all MODEL chips or only to one. Functions in the low level library are called from functions in the interface library, which are called itself from the application program and hide the system architecture from the application programmer.

Performance

A MODEL chip calculates one pairwise force within 400ns. If only a few MODEL chips are used the calculation time is proportional to $O(N^2)$. If there is a large number of MODEL chips in the system the calculation time of the nonbonded forces is shorter than the calculation time of the tasks running on the host. Then the simulation time is proportional to O(N), because it is limited by the computation time of the host computer and the communication time between the host computer and the MODEL chips.

An experiment with N=11,940 and 24 MODEL chips takes 3.6 seconds to advance one time step. With this the MD-Engine accelerates an MD calculation by a factor of 98 compared to a SPARCstation 10. The experiment shows that with a 6-card MD-Engine system, the accelerator is more or less optimized for a simulation systems with ~12,000 particles.

4.2 New Proposals

4.2.1 Parallel Gromos MD Algorithm

To reach the desired speed-up it is indispensable to calculate the solvent-solvent non bonded forces and the pairlist on the coprocessor. To maintain the full range of possible hardware architectures we need parallel models of these functions. The simulation space may be divided spatially into slices to distribute the force calculation on several processors. The number of slices equals the number of force processors. Two slices are not distinct because they overlap due to the cutoff radius. This distribution has the effect that the input data, the coordinates and the pairlist, must not be copied to all processors but are distributed according the spatial division. On the other hand, the bigger the cutoff radius gets compared with the slice width, the more data must be copied to the neighbouring force processor.

Fig. 4.15 shows a parallel version of the Gromos MD algorithm. The CDFG diagram in the figure is simplified to improve readability. We developed a generic model for the force calculation such that the number of force nodes is a parameter for other functions. We introduced new special functions like *fork* and *join*. These functions distribute and gather data, where the number of *dis* nodes is an input parameter for the *fork* function and the number of *sum* nodes an input parameter for the *join* node, respectively.

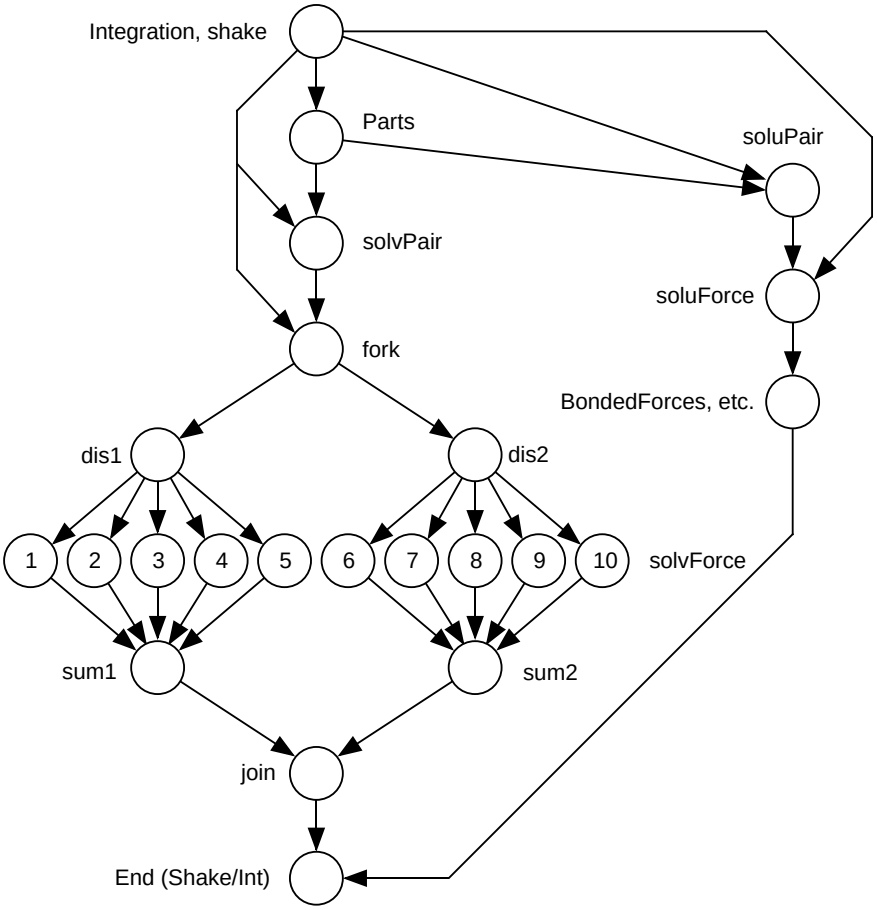


Fig 4.15 Simplified CDFG

The parallelisation of the pairlist function is more difficult, but with the division into *Parts*, *solvPair* and *soluPair*, in combination with a new efficient algorithm, an adequate approach

is guaranteed. The partial pairlist *solvPair* contains only solvent pairs where the *soluPair* list contains solute pairs and solute-solvent pairs.

The non bonded forces are split-up into one ore more solvent-solvent force calculation functions *solvForce* and one solute interaction function *soluForce*. The nodes *fork* and *dis* have no functional implementation, they distribute the data to the *solvForce* nodes. The nodes *sum* and *join* gather the results and calculate the partial intermediate sums of the forces of the charge groups located in the appropriate cutoff overlap zone.

4.2.2 Hardware Accelerator with General Purpose RISC Processors

How can this algorithm be mapped on a general parallel coprocessor with general purpose (RISC) micro processors? Let's take the benchmark with water and octahedron boundaries (table 3.3). If we want to accelerate the simulation time by a factor of ten on a today's workstation, we have 2.13 seconds per iteration to do all "outsourced" calculations and communications with the host, for a well balanced system. Thus, if the host computer performance doubles within the next 1.5 years the remaining time is about half a second, assumed our hardware is ready for delivery in two or three years. The problem is that the improvement in communication systems (bandwidth) is slower. What we try is to find an architecture fulfilling the computational requirements with the fastest available components (ASIC's, MCM, RISC processors) with an already today established communication system (e.g. PCI) for a host machine available in 3 years.

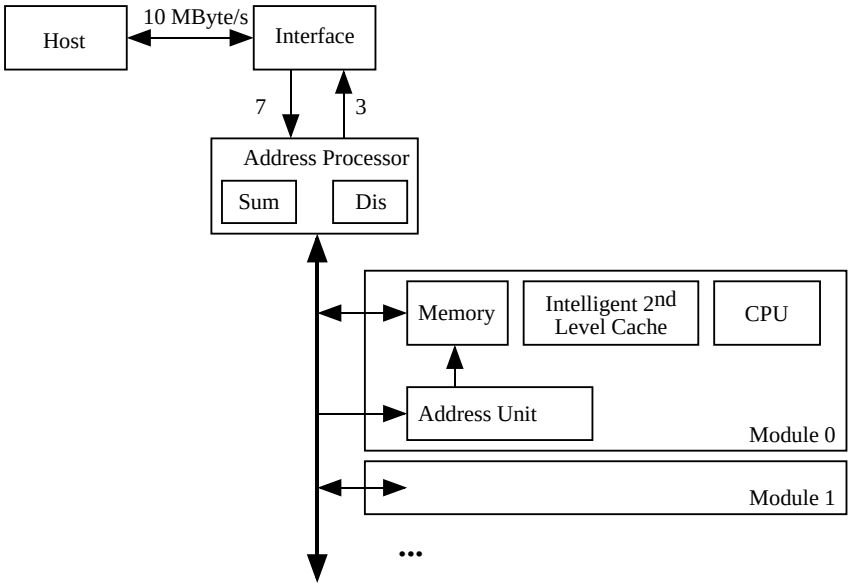


Fig 4.16 Coprocessor with general purpose RISC processors

Fig. 4.16 shows a possible architecture. With the models presented in paragraph 3.4, the profiling results, a speed-up of ten, the expected bandwidth needed between host and coprocessor lies around 10 Mbyte/s. A PCI bus allowing block transfers with a device driver supporting asynchronous DMA transfers on both sides provides enough bandwidth. On the coprocessor side, we are free to design very fast data connections (64 bit bus, high speed serial links) and intelligent data distribution and collector processors (ASIC's). Calculation is done with the newest and fastest RISC processors (ALPHA, PowerPC), connected to an application specific memory structure allowing shortest access times or data pipelining.

The address processor acts as a bus master on coprocessor side distributing data and collecting and adding up the partial forces to keep the number of data to transfer over the host interface low.

4.2.3 Hardware with Sharc Signal Processor

Analog Device's SHARC (Super Harvard ARchitecture Computer) is a high performance 32-bit digital signal processor. The SHARC builds on the ADSP-21000 family DSP core to form a complete system on-a-chip, adding a dual-port on-chip SRAM and integrated I/O peripherals supported by a dedicated I/O bus. Four independent buses for dual data, instructions, I/O, plus crossbar switch memory connections, comprise the super Harvard architecture. It has a clock-frequency of 40 MHz, which results in a sustained performance of 40 Mips and a peak performance of 120 Mflops. The block diagram of the SHARC DSP is shown in Fig. 4.17.

The ADSP-2106x core processor consists of three independent computation units: an ALU, a multiplier with a fixed-point accumulator and a shifter. The computation units process data in three formats: 32-bit fixed-point, 32-bit floating-point and 40-bit floating-point. Floating-point formats are compatible to standard IEEE format. The ALU performs a standard set of arithmetic and logic operations in fixed- and floating-point formats. The multiplier performs fixed- and floating-point multiplication as well as fixed-point multiply/add and multiply/subtract operations. The shifter performs logical and arithmetic shifts, bit manipulation, field deposit, extraction and exponent derivation operations on 32-bit operands. The computation units perform single-cycle operations without being pipelined. There can be a multifunction computation where the ALU and multiplier perform independent, simultaneous operations.

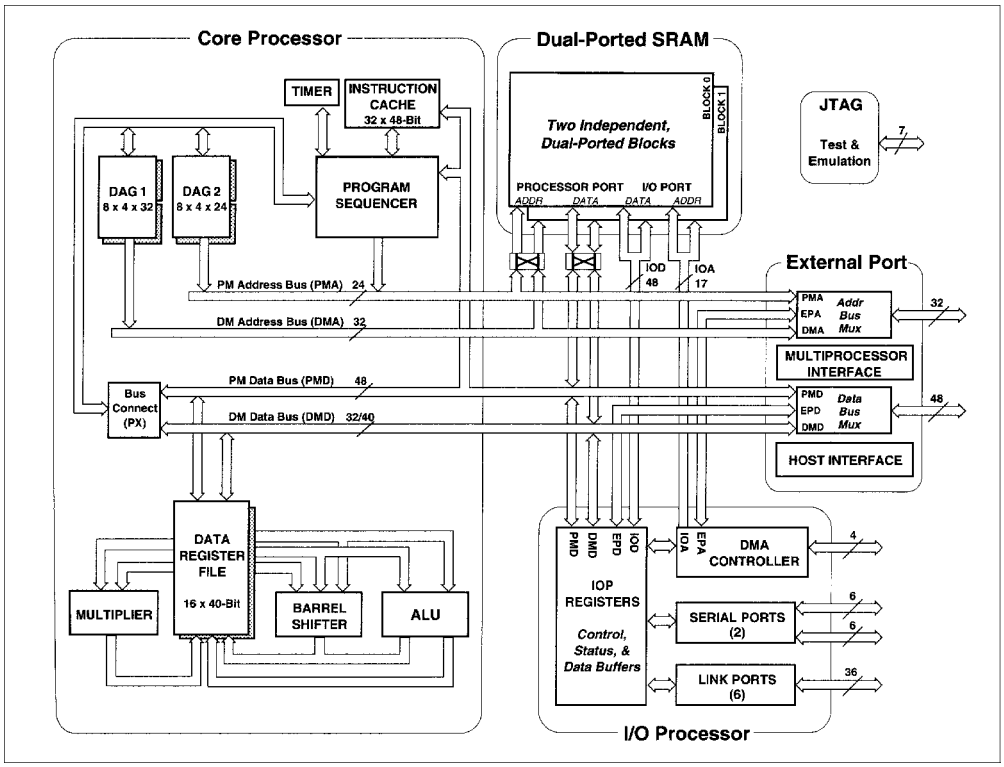


Fig 4.17 Block diagram of SHARC DSP

The data register file contains two sets of sixteen 40 bit registers, to allow for fast context switching. It is used for transferring data between the computation units and the data buses and for storing intermediate results.

The program sequencer supplies instruction addresses to the program memory. It controls loop iterations and evaluates conditional instructions. The ADSP-2106x achieves its fast execution rate by means of pipelined *fetch*, *decode* and *execute* cycles. With its instruction cache, it can simultaneously fetch an instruction from cache and access two data operands from memory.

The data address generators (DAGs) provide memory addresses when data is transferred between memory and registers. Dual data address generators enable the processor to simultaneous output any addresses for two operand reads or writes. DAG1 supplies 32-bit addresses to data memory. DAG2 supplies 24-bit addresses to program memory for program memory access. Each DAG keeps track of up to eight address pointers, eight modifiers and eight length values. A length value may be associated with each pointer to perform automatic modulo addressing for circular data buffers. With an internal loop counter and loop stack, the ADSP-2106x executes loop code with zero overhead. No explicit jump instructions are required to decrement and test the counter. The DAGs automatically handle address pointer wraparound, reducing overhead, increasing performance, and simplifying implementation. Each DAG register has an alternate register that can be activated for fast context switching.

The processor core has four buses: program memory address bus, data memory address bus, program memory data bus, and data memory data bus. The data memory stores data operands while the program memory is used to store both instructions and data. This allows for dual data fetches, when the instruction is supplied by the instruction cache. The PM Address bus is 24 bits wide, allowing access of up to 16 Mwords of mixed instructions and data. The PM Data bus is 48 bits wide to accommodate the 48-bit instruction width. The DM Address bus is 32 bits wide allowing direct access of up to 4 Gwords of data. The DM data bus is 40 bits wide. The PX bus connect registers permit data to be passed between the 48-bit PM data bus and the 40-bit DM Data bus or between the 40-bit register file and the PM Data bus.

The ADSP-21060 contains 4 Mbits of on-chip SRAM, organized as two blocks of 2 Mbits each, which can be configured for different combinations of program and data storage. The ADSP-21062 includes 2 Mbit SRAM, organized as two 1 Mbit blocks. Each memory block is dual-ported for single-cycle, independent access by the core processor and I/O processor or DMA controller. The memory can be configured as a combination of different word sizes up to 4 or 2 Mbits. All of the memory can be accessed as 16-bit, 32-bit or 48-bit words.

The external port provides the processor's interface to off-chip memory and peripherals. The four Gword off-chip address space is included in the ADSP-2106x's unified address space. The separate on-chip buses are multiplexed at the external port to create an external system bus with a single 32-bit address bus and a single 48-bit data bus. External SRAM can be either 16, 32, or 48 bits wide. The ADSP-2106x's on chip DMA controller automatically packs external data into the appropriate worth width. Separate control lines allow for simplified addressing of page-mode DRAM. The ADSP-2106x provides programmable memory wait states and external memory acknowledge controls to allow interfacing to DRAM and peripherals with variable access, hold and disable time requirements.

The host interface allows easy connection to standard microprocessor buses with little additional hardware required. It is accessed through the external port and is memory mapped into the unified address space. The host can directly read and write the internal memory of the ADSP-2106x.

The ADSP-2106x has two synchronous serial ports that provide an inexpensive interface to a wide variety of digital and mixed-signal peripheral devices. It can operate at the full clock rate of the processor. Serial port data with word lengths selectable from 3 to 32 bits, can be automatically transferred to and from on-chip memory via DMA. The ADSP-2106x has six 4-bit link ports, which can be clocked twice per cycle allowing both to transfer 8 bits per cycle. Link port I/O is especially useful for point-to-point interprocessor communication in multiprocessing systems. The link port can operate independently and simultaneously, with a maximum data trough-put of 240 Mbytes/s. Data is packed into 32-bit or 48-bit words and can be directly read by the core processor or DMA-transferred to on-chip memory.

The DMA controller allows zero-overhead data transfers without processor intervention. It operates independently and invisibly to the processor core. DMA transfers can occur between the ADSP-2106x's internal memory and external memory, external peripherals, or a host processor. DMA transfers can also occur between the ADSP-2106x's internal memory and its serial port or link ports. Ten channels of DMA are available on the ADSP-2106x; two over the link ports, four over the serial ports and four over the processor's external port. Therefore code and data transfers can be accomplished with low software overhead.

Up to six ADSP-2106xs and a host processor can directly be connected together. Distributed bus arbitration logic is included on chip. Bus arbitration is selectable as either fixed or rotating priority. The unified address space allows direct interprocessor access of each ADSP-2106x's internal memory. Master processor change-over requires only one cycle of overhead. Maximum throughput for interprocessor data transfer is 240 Mbytes/s over the link ports or external port. Broadcast writes allow simultaneous transmission of data to all ADSP-2106xs.

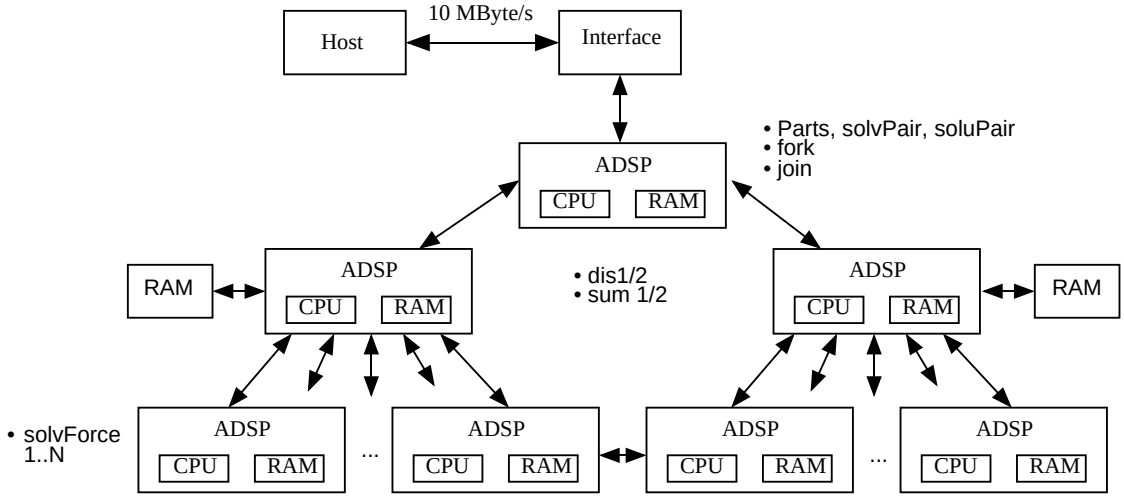


Fig 4.18 Hierarchical sharc coprocessor board

We implemented the distance algorithm on an ADSP-21060 and have optimized and simulated the programming code. A benchmark is shown in paragraph 6. The ADSP-2106x is described in more details in Analog Devices SHARC documentation [28].

To map the Gromos functions on a SHARC architecture, we suggest a hierarchical architecture as in fig. 4.18: The connections between the SHARC's are high speed link ports, the host connection may be a PCI bus to provide enough bandwidth.

The bulleted function-list near the processors in Fig. 4.18 relates to those in Fig. 4.15.

4.3 Comparison

4.3.1 Existing Third-Party Solutions

MD-Grape calculates forces and viral directly. For energies two runs are necessary. It could be used in force- or potential mode. DFT/IDFT mode is not necessary for GROMOS. With MD-Grape you have a problem to generate a pairlist, because distances are stored internally in the MD-chip and cannot be obtained externally. Position vectors are stored in 40-bit fixed point format. This is more precise than the 32-bit floating point format but the number range is not so high as used for GROMOS where coordinates are stored in 32-bit floating point format. Therefore MD-Grape is not suitable to implement the GROMOS algorithm.

GRAPE-4 is huge system, on which one could compute all the necessary calculations, which should be made on our coprocessor. But GRAPE-4 is not suited as coprocessor because of its size. In addition it is quite expensive. The GROMOS-coprocessor should be small box, which can be placed close to a workstation and which should not be too expensive.

The GROMACS system is built similar as our coprocessor system could be. Only non-bonded forces are calculated on GROMACS and the calculation space is splitted into boxes for calculating the neighbour list as we may consider for our pairlist calculation. Particles data distribution and communication over the ring is a good idea to solve the N^2-problem. We will probably use a ring architecture too, if our coprocessor system is a parallel processor, but we will need to use then faster processor elements.

There is a similar problem to use the MD-Engine as for GRAPE-4: it is a large, expensive system. The most interesting thing about MD-Engine is the MODEL-chip, with which we could calculate the forces, virial and energies of the atoms.

4.3.2 New Proposals

Both suggestions are scalable architectures. This is important to adapt the performance of the coprocessor system to the simulation complexity. The major advantages of the SHARC system are the simplicity of assembly, the availability of a C compiler for the processors including third party libraries and drivers, the disadvantage is the need to program in assembler to get efficient implementations. The advantage of the RISC system is the availability of excellent compilers so there is no need to program in assembler. This fact leads to portable and therefore reusable function specifications and implementations. The need of lots of user specified hardware, ASIC's, and the general heterogeneity of the system are the disadvantages.

5 Hardware Modelling

5.1 Communication Models for Host and Sharc

Our target system will be a kind of distributed memory multiprocessor (host and coprocessor system) in which processors communicate with each other by point to point links. Our communication model describes the performance data of the underlying interconnection network, but not the physical layer itself. We defined the following main communication parameters:

L latency: time to communicate one single word from the source processor to a destination memory of another processor.

o overhead: time that a processor is occupied in the transmission or reception of data. During this time, the processor cannot perform other operations.

g gap: minimum time between two consecutive writes or reads on a channel. The reciprocal of g is the bandwidth available on this channel.

We used this communication model to model the MD algorithm and the hardware (paragraph 6). We set L=0 and g=0 for host internal and sharc internal communication. The host-coprocessor interface is specified with g=0.1sec/Mbyte, the sharc-sharc communication according to the data sheet (link ports) with g=0.025sec/Mbyte.

With a first approximation the latency and overhead was neglected (o=0 in Fig. 5.1). It was assumed that communication is done in the background without slowing down the processor. In addition, an overlap scheme is introduced enabling the subsequent operation to start even if the communication is not finished.

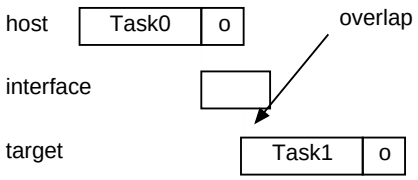


Fig 5.1 Overlap

5.2 Performance Models for Host and Sharc

The pairlist and solvent force calculation routines were tested on different host machines to compare the specified performance (common used benchmarks on workstations) with the real performance achieved for our functions. We made use of eqn (3.7) and eqn (3.9), combined with the measured time, and get 41 Mflops for the Sun (Sun ultra 1 creator, 170 MHz, specification from sun: SPECint_95: 7.44, SPECfp_95: 10.4). The SHARC could not yet be measured, we choose an average of 80 Mflops, what seems suitable for functions directly programmed in assembler.

5.3 Generally Applicable Multiprocessor Models

In this chapter we examine speed-up and communication overhead of different parallel processor topologies with respect to the number of processing elements (PEs). The architectures of the examined systems are distributed memory systems. Each PE consists of a processor, a memory block and a communication node, which interconnects the PE and the communication net. The goal was to find a possible data distribution among PEs such that data can be exchanged without or with only few collisions. Similar topics are treated in [12] and [15].

5.3.1 General Calculation Problem

As simple example we use the solution of the $O(N^2)$ pairlist calculation problem. The calculation of distances, forces, energies and virial in GROMOS is always done between pairs of atoms. The pairlist is generated by calculating the distances between all molecules. Therefore the acceleration of the distance calculation is an important part of the algorithm. The results of the distance calculation can then be placed in a matrix (Fig. 5.2), where the molecules (molecule number) specifies the x-y coordinate. The matrix is symmetric and always one of the two corresponding entries is positive, the other negative. Therefore only half of the matrix has to be calculated and can then be mirrored and multiplied with -1. In Fig. 5.2 only the parts in the upper triangle must be calculated and can then be mirrored into the hatched part, which has to be multiplied with -1.

For our investigations only the solvent-solvent part is considered on the GROMOS coprocessor.

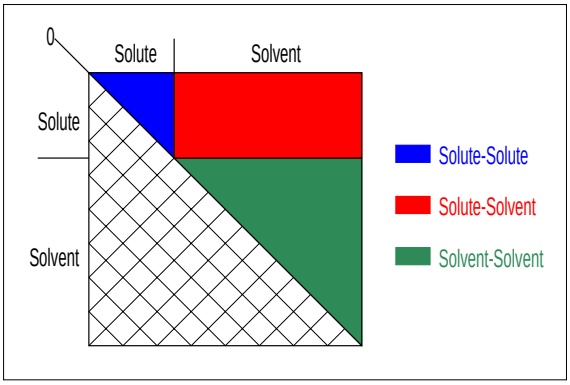


Fig 5.2 Calculation matrix

To determine which molecules interact, the question is how to distribute the molecules among the PEs. Assuming that all PEs are identical, the best thing would be to distribute all molecules evenly onto PEs to save memory space. With this, there is the problem that the first molecule has to be compared with all others, the second must be compared with all others except the first, the third must be compared with all others except the first and the second and so on (Fig. 5.3). This complicates communication or requires uneven distribution of molecules among PEs.

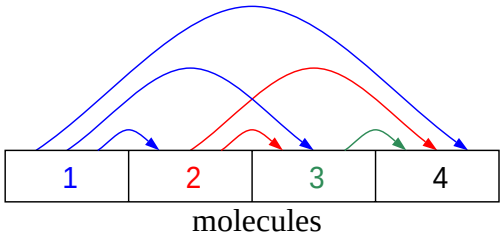


Fig 5.3 Tie of the molecules

In the following chapters different communication topologies and data exchange methods between PEs are described. We have estimated the communication overhead with the number of PEs and molecules and have compared the different topologies.

The communication time to transfer data from the host to the coprocessor is neglected in the estimations. Only the number of computation steps and communication cycles on the coprocessor are taken into account. In one computation step two molecules per processor are compared. As further simplification we assume, that a processor is able to calculate the distance of one molecule-pair within one clock cycle. This is e.g. made possible with pipelining, where in each clock cycle a new pair of coordinates is read and a distance is transferred to memory.

Each processor can access the memory of any other PE through its communication node. As memory we assume dualported SRAMs. With this, the communication node can access data without interrupting the processor. We have modelled the communication node, such that it can send data to a PE and simultaneously request data from another PE.

The number of communication cycles is determined by the communication latency. The latency is measured in clock cycles. We assume the data width such, that one of the three coordinates of an atom can be sent within one cycle. Therefore three communication cycles are needed to transfer the coordinates of one atom or one molecule.

Molecules, are only stored in the memory of one single PE. Locally not available molecules are requested through the communication net from the corresponding PE and are immediately used by the requesting processor for computation. The result is stored in the local memory of the PE where it was calculated and the requested molecule is discarded. We compare our estimation with an ideal speed-up, where communication time is 0 and load is evenly balanced on all PEs. The computation time on an ideal parallel processor is:

$$T = \frac{1}{P}\left(\sum_{i=1}^{NSM-1} i\right) = \frac{1}{2P}(NSM-1)\cdot NSM \qquad (5.1)$$

where P is the number of PEs and NSM is the total number of molecules.

5.3.2 Bus Architecture

A bus is the simplest architecture but also implies the biggest communication overhead, if the molecules are equally distributed on the PEs

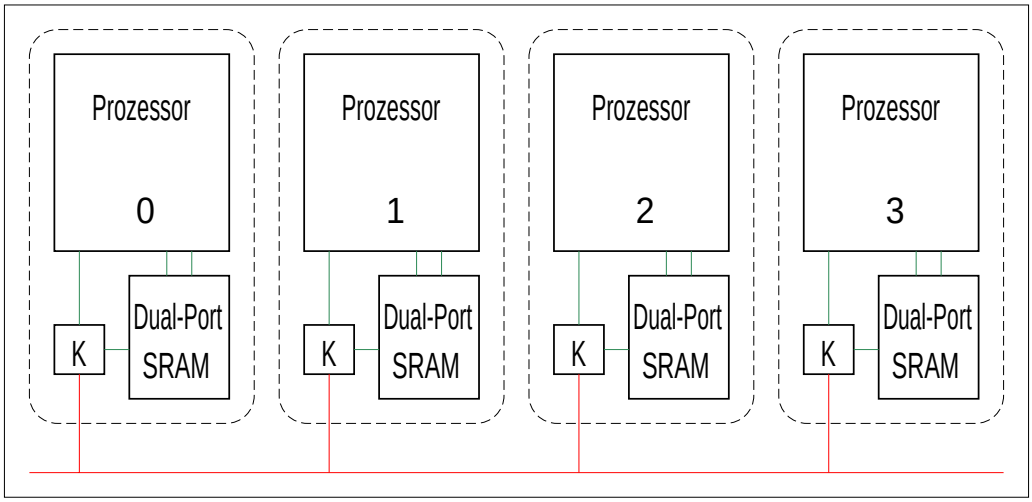


Fig 5.4 Bus Architecture

For this case it is not possible having computation and communication in parallel, therefore we have not further analysed this case.

Here the best thing is to store all molecules on all PEs. With this, the molecules must be distributed over all PEs at the beginning of the calculation, therefore the bus is not used during computation. This model allows for almost linear speed-up with the number of processors but at the cost of memory.

Because there is no communication during computation the communication time is 0 as for the ideal speed-up. The computation time for a bus system is:

$$T = \mathrm{Round}\left(\frac{1}{P}\sum_{i=1}^{NSM-1} i\right) + 1 = \mathrm{Round}\left(\frac{1}{2P}(NSM-1)\cdot NSM\right) + 1 \qquad (5.2)$$

5.3.3 2D-Net

With a 2D-net including diagonal connections every processor can simultaneous read data from any other neighbouring PE's memory, see fig. 5.5. All communication links are assumed to be full duplex.

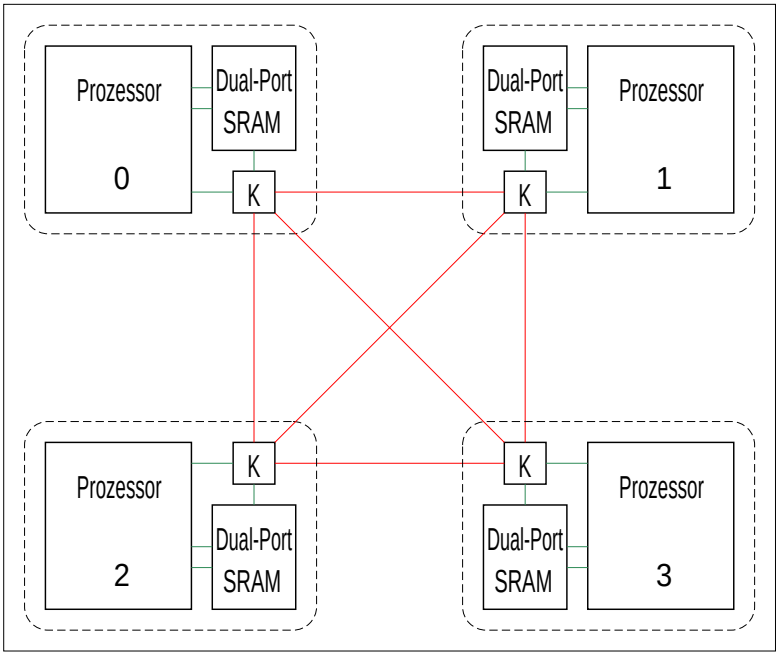


Fig 5.5 2D-net with 6 communication links

If we look at the calculation problem in paragraph 5.3.1 we see, that every processor must first compare all molecules in its local memory. Then all molecules from its local memory must be compared with all molecules from the other PEs. To avoid double calculations, every processor must only compare half of the molecules in its local memory with all molecules on the other PEs to avoid double calculating. For a better understanding, we split the molecules in a PE's memory into two sets A and B. Each set of every PE must be compared with all other sets of the same and the other PEs.

With this a computation scheme can be found, that implies no communication conflicts. The table below shows, how calculation is done and from which memory each processor takes the molecules:

Timestep	P0	P1	P2	P3
1	0A - 0A	1A - 1A	2A - 2A	3A - 3A
2	0B - 0B	1B - 1B	2B - 2B	3B - 3B
3	0A - 0B	1A - 1B	2A - 2B	3A - 3B
4	0A - 1A	1B - 0B	2A - 3A	3B - 2B
5	0A - 1B	1A - 0B	2A - 3B	3A - 2B
6	0A - 2A	1A - 3A	2B - 0B	3B - 1B
7	0A - 2B	1A - 3B	2A - 0B	3A - 1B
8	0A - 3A	1A - 2A	2B - 1B	3B - 0B
9	0A - 3B	1A - 2B	2A - 1B	3A - 0B

Table 5.1 Computation scheme with 4 processor elements and 6 full duplex links

We see, that the available communication bandwith is not fully utilized in table 5.1. There are 6 full duplex links. With this 12 data sets can be exchanged simultaneously among PEs. In the above computation scheme only 4 connections are used simultaneously. If we only allow for half duplex links, the following computation scheme can be found.

Timestep	P0	P1	P2	P3
1	0A - 0A	1A - 1A	2A - 2A	3A - 3A
2	0B - 0B	1B - 1B	2B - 2B	3B - 3B
3	0A - 0B	1A - 1B	2A - 2B	3A - 3B
4	0A - 1A	1B - 3B	2B - 0B	3A - 2A
5	0A - 2A	1B - 0B	2B - 3B	3A - 1A
6	0A - 1B	1A - 2B	2A - 3B	3A - 0B
7	0A - 3B	1A - 0B	2A - 1B	3A - 2B
8	0A - 2B	1B - 3A	2A - 1A	3B - 0B
9	0A - 3A	1B - 2B	2A - 0B	3B - 0B

Table 5.2 computation scheme with 4 processor elements and 6 half duplex links

Also here only 4 communication connections are used simultaneously. The results in table 5.2 also show, that in the first three time steps no connections are utilized. During the 9 timesteps only 24 connections are set up in total. Without diagonal communication links, two adjacent communication links must be used to support one diagonal transaction. Since there are 8 diagonal transactions, we need 24+8=32 connections for the whole calculation.

Table 5.3 shows the computation scheme when only 4 half duplex links are available.

With the 2D-net and 6 communication links, load balancing is regular in each timestep. If we only have 4 links, load balancing is not regular. In table 5.3, we see that at timestep 3 processors 0 and 3 calculate molecules both within the second half of their local memory and where processors 1 and 2 calculate molecules from their local and from other memories. Therefore processors 1 and 2 have to calculate more interactions than processors 0 and 3 in timestep 2. In timestep 4 its the same but vice versa.

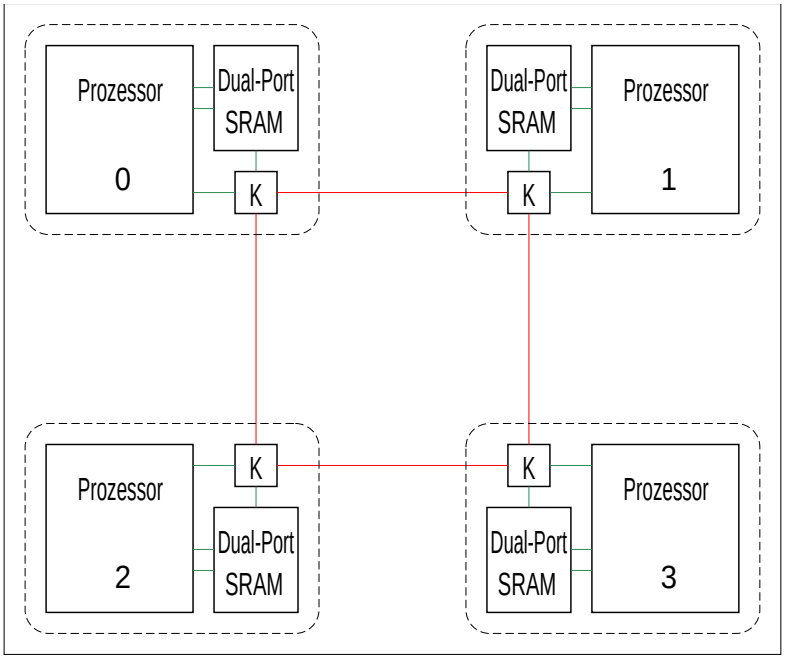


Fig 5.6 2D-net with four communication links

Timestep	P0	P1	P2	P3
1	0A - 0A	1A - 1A	2A - 2A	3A - 3A
2	0A - 0B	1A - 2A	2B - 1B	3A - 3B
3	0B - 0B	1A - 2B	2A - 1B	3B - 3B
4	0A - 3A	1B - 1B	2B - 2B	3B - 0B
5	0A - 3B	1A - 1B	2A - 2B	3A - 0B
6	0A - 1A	1B - 3B	2B - 0B	3A - 2A
7	0A - 1B	1A - 3B	2A - 0B	3A - 2B
8	0A - 2A	1B - 0B	2B - 3B	3A - 1A
9	0A - 2B	1A - 0B	2A - 3B	3A - 1B

Table 5.3 Computation scheme with 4 processor elements and 4 half duplex links

The communication links are only used, when a new molecule from another PE has to be fetched. After that, the communication net is not used. Therefore processors must not be halted, if load balancing is uneven at a timestep. They can already begin to calculate the interactions for the next timestep. In this way, load balancing over the whole computation gets regular, because all processors have to calculate the same number of interactions during the whole computation. The calculation time for a 2D net can be computed as follows:

$$T_{cal} = \left(\sum_{i=1}^{\frac{NSM}{P}-1} i \right) + \frac{1}{2} \cdot \frac{NSM}{P} \cdot \left(1 - \frac{1}{P}\right) NSM \qquad (5.3)$$

For the communication time only the number of molecules must be considered, which are transferred through the communication net. These number has to be multiplied with the communication latency *l*: The communication time in a 2D-net results in:

$$T_{com} = l \cdot \left(1 - \frac{1}{P}\right) NSM \tag{5.4}$$

The whole computation time for the 2D net is:

$$T = T_{cal} + T_{com} \tag{5.5}$$

$$T = \frac{1}{2}\left(\frac{NSM}{P} - 1\right)\left(\frac{NSM}{P}\right) + \left(\frac{NSM}{2P} + l\right)\left(1 - \frac{1}{P}\right) NSM \tag{5.6}$$

A 2D-net supports up to four PEs. If more need be used, the communication network will become a bottleneck. In the next section we examine communication nets, which support more processors.

5.3.4 Hyper Cube

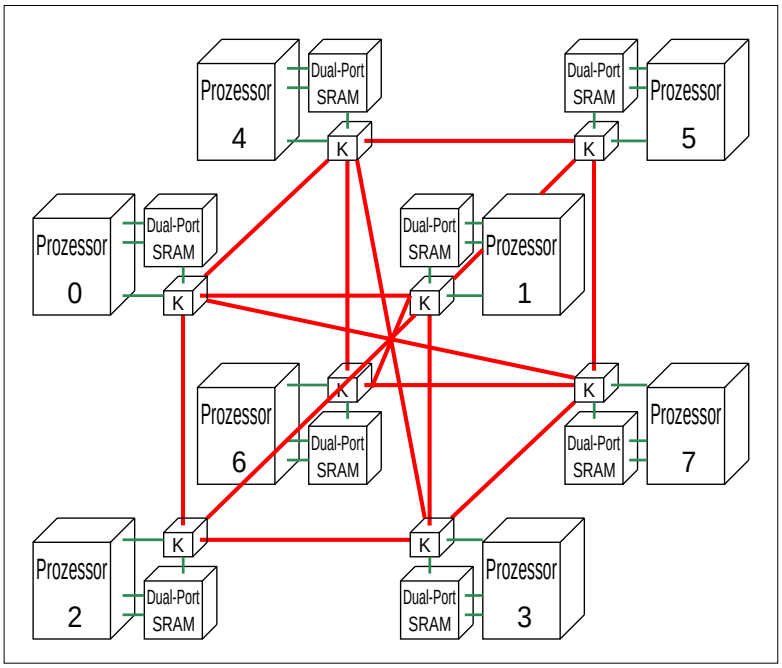


Fig 5.7 Hyper cube (3D-net)

In a hyper cube only the PEs on the edges at one side are directly connected together. Transfers of data from across the diagonal have to be routed through another PEs. As for the 2D-net the communication links are assumed to be half duplex.

Let's consider a system with 8 PEs. The molecules are divided into two sets again as in paragraph 5.3.3. One set on a processor has to be compared with all other sets on the other PEs, which takes 7*2=14 timesteps. In addition each processor has to compare all molecules within its local memory. This takes 3 timesteps since each of the two sets have to be compared with itself and with each other. Therefore it takes 17 timesteps to calculate all interactions.

We can now calculate the required connections. 17 timesteps and 12 communication links are available. This results in totally 204 available connections for the entire computation. It takes two communication links at a time for fetching data from a diagonally placed PE on

the same plane and three communication links for fetching data from a PE, which is placed diagonally across the cube. Determining the required links, we get 8*(2*3*1 + 2*3*2 + 2*1*3)=192 connections for the entire computation.

It should theoretically be possible, to develop a computation method without communication collisions. But the problem is that 192/17=11.3, which means, that in every timestep all communication links must be used. But no computation method could be found, which can utilize all 12 communication links.

As stated above 8 PEs are available and on the average 8 connections can be used at time. Thus 136 connections are available within 17 timesteps. This is to few for communication without conflicts. That's why we have added four diagonal communication links as shown in Fig. 5.7. Thus reduces the required number of connections to 8*(2*3*1 + 2*3*2 + 2*1*1)=80 for the entire computation. Table 5.4 shows the computation scheme. As in table 5.3 the load balancing is not equal in all timesteps, but processors need not to be halted for the same reason as discussed in paragraph 5.3.3, thus making the load balancing even over the whole computation.

Timestep	P0	P1	P2	P3	P4	P5	P6	P7
1	0A - 0A	1A - 1A	2A - 2A	3A - 3A	4A - 4A	5A - 5A	6A - 6A	7A - 7A
2	0A - 0B	1A - 4A	2A - 2B	3A - 6A	4B - 1B	5A - 5B	6B - 3B	7A - 7B
3	0B - 0B	1A - 4B	2B - 2B	3A - 6B	4A - 1B	5B - 5B	6A - 3B	7B - 7B
4	0A - 5A	1B - 1B	2A - 7A	3B - 3B	4B - 4B	5B - 0B	6B - 6B	7B - 2B
5	0A - 5B	1A - 1B	2A - 7B	3A - 3B	4A - 4B	5A - 0B	6A - 6B	7A - 2B
6	A0 - 2A	1B - 0B	2B - 3B	3A - 1A	4B - 5B	5A - 7A	6A - 4A	7B - 6B
7	0A - 2B	1A - 0B	2A - 3B	3A - 1B	4A - 5B	5A - 7B	6A - 4B	7A - 6B
8	0A - 4A	1B - 3B	2B - 0B	3A - 7A	4B - 6B	5A - 1A	6A - 2A	7B - 5B
9	0A - 4B	1A - 3B	2A - 0B	3A - 7B	4A - 6B	5A - 1B	6A - 2B	7A - 5B
10	0A - 1A	1B - 6B	2A - 4A	3A - 5A	4B - 2B	3B - 5B	6A - 7A	7B - 0B
11	0A - 1B	1A - 6B	2A - 4B	3A - 5B	4A - 2B	3B - 5A	6A - 7B	7A - 0B
12	0A - 3A	1B - 5B	2B - 6B	3B - 0B	4A - 7A	5A - 2A	6A - 1A	7B - 4B
13	0A - 3B	1A - 5B	2A - 6B	3A - 0B	4A - 7B	5A - 2B	6A - 1B	7A - 4B
14	0A - 6A	1A - 7A	2B - 5B	3A - 2A	4B - 3B	5A - 4A	6B - 0B	7B - 1B
15	0A - 6B	1A - 7B	2A - 5B	3A - 2B	4A - 3B	5A - 4B	6A - 0B	7A - 1B
16	0A - 7A	1A - 2A	2B - 1B	3A - 4A	4B - 0B	5A - 6A	6B - 5B	7B - 3B
17	0A - 7B	1A - 2B	2A - 1B	3A - 4B	4A - 0B	5A - 6B	6A - 5B	7A - 3B

Table 5.4 Computation scheme with 8 processors in a hyper cube

Since the same number of molecules are transferred through the communication net as with the 2D-net, the computation time of the hyper cube can also be calculated with eqn (5.3), eqn (5.4), eqn (5.5) and eqn (5.6).

An extension of a hyper cube to more processors is difficult. With 16 processors you would have a 4D-net, with 32 processors you need a 5D-net and so on. It is expensive to physically realize such a system.

5.3.5 Recursive Structure

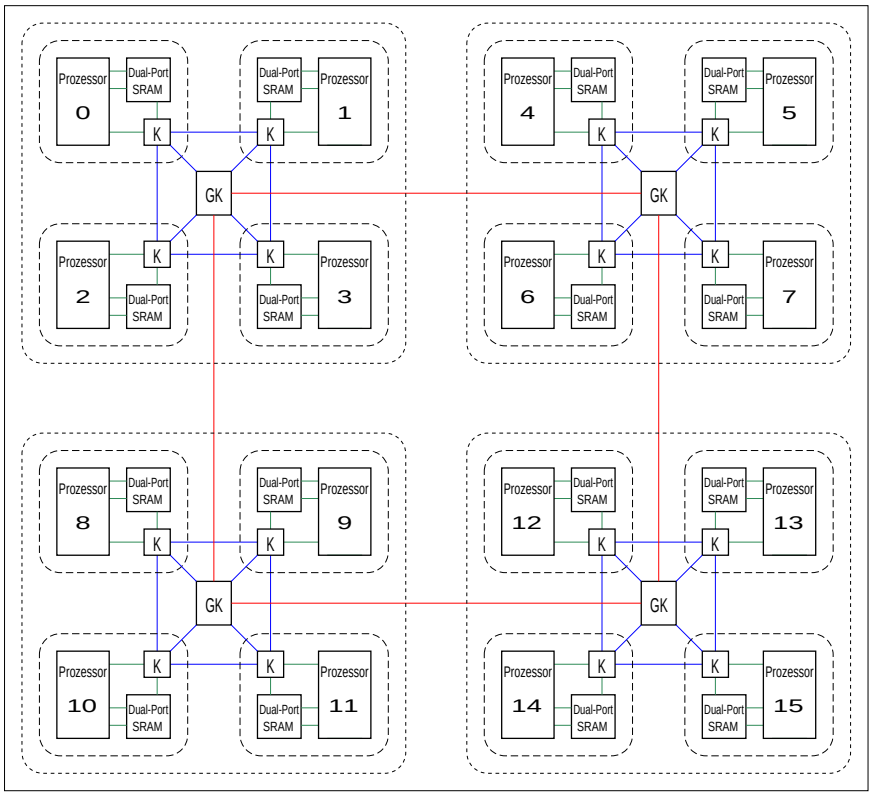


Fig 5.8 Recursive structure

A multiple processor architecture could e.g. be built recursive from a 2D system as shown in fig. 5.8. In this approach one PE is a cluster with 4 PEs. The computation method on the PEs in a cluster is the same as in table 5.3 of the 2D-net. Additionally each communication node is connected to a global node, which are itself connected to the global communication net, on which again the same computation method is used as on the processor clusters.

Table 5.5. shows a possible computation scheme. It is a recursive extension of table 5.3 of the 2D-net and is simplified to improve readability. Only the number of the processor memory from which the second particle is fetched, is listed, because the first molecule is always fetched from the own processor memory. Remember: each of the processor calculates only half of the interactions from the memories, this is not listed in the table because of the complexity.

With the recursive structure the calculation time remains the same as with the other structures:

$$T_{cal} = \left(\sum_{i=1}^{\frac{NSM}{P}-1} i \right) + \frac{NSM}{2P} \cdot \left(1 - \frac{1}{P}\right) NSM \qquad (5.7)$$

Concerning the communication time we have now to consider two different latencies. One is the local latency l_l for fetching data in the same processor cluster over the local net and the second is the global latency l_g for fetching data from another processor cluster over the global nodes and global communication net, which takes more time.

For the communication time in a recursive structure follows

$$T_{com} = l_l \cdot \left(\frac{1}{Q} - \frac{1}{P}\right) NSM + l_g \cdot \left(1 - \frac{1}{Q}\right) NSM \tag{5.8}$$

where *P* stands for the number of processors and *Q* is the number of processor clusters.

The total computation time for a recursive structure is

$$T = T_{cal} + T_{com} \tag{5.9}$$

$$T = \frac{1}{2}\left(\frac{NSM}{P} - 1\right)\left(\frac{NSM}{P}\right) + \left(\frac{NSM}{2P}\left(1 - \frac{1}{P}\right) + l_l\left(\frac{1}{Q} - \frac{1}{P}\right) + l_g\left(1 - \frac{1}{Q}\right)\right) NSM \tag{5.10}$$

The advantage of this communication structure is, that it can simply be extended up to a system with 64 processors. In this case, the global structure would be a hyper cube and every processor cluster would be again a hyper cube itself. A system with 32 processors could either be a 2D-net with hyper cube processor clusters or a hyper cube with 2D-net processor clusters. Concerning communication time the first is better, since only 3/4 of the communication is done over the global net compared to 7/8 as in the second case.

The recursive structure can be refined into more recursive steps so that you have for example a 2D-net global net with 4 processor systems, every processor system itself being a hyper cube with 8 processors clusters and every processor cluster is a 2D-net again with 4 processor elements, but the communication latency would increase to an unacceptable level.

Time-step	**P 0**	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**	**P 6**	**P 7**	**P 8**	**P 9**	**P 10**	**P 11**	**P 12**	**P 13**	**P 14**	**P 15**
1	0	2	3	1	8	8	8	8	12	12	12	12	4	4	4	4
2	3	1	0	2	9	9	9	9	13	13	13	13	5	5	5	5
3	1	3	2	0	10	10	10	10	14	14	14	14	6	6	6	6
4	2	0	1	3	11	11	11	11	15	15	15	15	7	7	7	7
5	12	12	12	12	4	6	7	5	0	0	0	0	8	8	8	8
6	13	13	13	13	7	5	4	6	1	1	1	1	9	9	9	9
7	14	14	14	14	5	7	6	4	2	2	2	2	10	10	10	10
8	15	15	15	15	6	4	5	7	3	3	3	3	11	11	11	11
9	4	4	4	4	12	12	12	12	8	10	11	9	0	0	0	0
10	5	5	5	5	13	13	13	13	11	9	8	10	1	1	1	1
11	6	6	6	6	14	14	14	14	9	11	10	8	2	2	2	2
12	7	7	7	7	15	15	15	15	10	8	9	11	3	3	3	3
13	8	8	8	8	0	0	0	0	4	4	4	4	12	14	15	13
14	9	9	9	9	1	1	1	1	5	5	5	5	15	13	12	14
15	10	10	10	10	2	2	2	2	6	6	6	6	13	15	14	12
16	11	11	11	11	3	3	3	3	7	7	7	7	14	12	13	15

Table 5.5. Computation scheme with 16 processors in a recursive communication structure

5.3.6 Ring

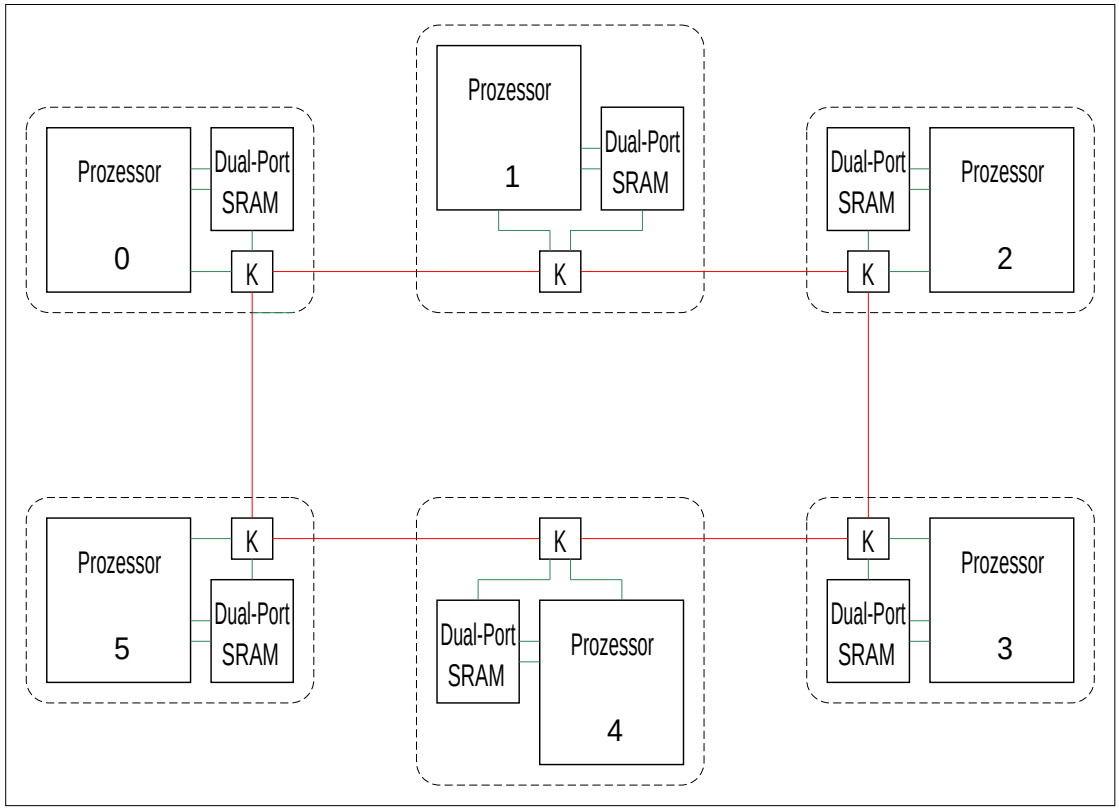


Fig 5.9 Ring structure

For a ring structure a different approach for the computation method has to be taken. It is not possible to transfer data directly from one PE to another PE without intermediate steps, if they are not adjacent. But if we look at the computation schemes of the other structures, we see, that for example the second part of the molecules of PE 0 is sent to all other PEs. Because the molecules are temporarily stored on the PEs, on which they are calculated, they need not to be sent directly from PE 0 to one of the other PEs at the beginning of each time-step, they can directly be forwarded from the PE, on which they are calculated, to the next PE.

Table 5.6 shows this computation method. All molecules are sent directly from the preceding PE in the timestep above. The scheme is made with the structure of Fig. 5.9. The molecules of a processor element must only be sent over half of the ring. In table 5.6 memory are still splitted into two parts and every part is sent separately over the ring. A better method is to send all molecules together over half of the ring. If the number of processors is even, only half of the molecules have to be compared in the last timestep because the two processors, which are opposite over the ring have the molecules from each other. E.g. in Fig. 5.9 there are processors 0 and 3. If the number of processors is odd, the molecules of the processors have to be only sent over the half of the ring minus one and then all molecules are compared in the last timestep.

The calculation time is the same as with eqn (5.3) and the communication time is the same as in eqn (5.4). With this the computation time is the same as for the 2D-net, see eqn (5.5). The cost of the circuit are much lower for a ring compared to the another structures, because data on the ring must be sent only in one direction and therefore all links are unidirectional.

In this case the communication time is shorter than in the other structures. Another advantage of this structure is the simple expansion possibility. The ring can be easily expanded with more processors.

Timestep	P0	P1	P2	P3	P4	P5
1	0A - 0A	1A - 1A	2A - 2A	3A - 3A	4A - 4A	5A - 5A
2	0B - 0B	1B - 1B	2B - 2B	3B - 3B	4B - 4B	5B - 5B
3	0A - 0B	1A - 1B	2A - 2B	3A - 3B	4A - 4B	5A - 5B
4	0A - 5A	1A - 0A	2A - 1A	3A - 2A	4A - 3A	5A - 4A
5	0B - 5A	1B - 0A	2B - 1A	3B - 2A	4B - 3A	5B - 4A
6	0A - 4A	1A - 5A	2A - 0A	3A - 1A	4A - 2A	5A - 3A
7	0B - 4A	1B - 5A	2B - 0A	3B - 1A	4B - 2A	5B - 3A
8	0A - 3A	1A - 4A	2A - 5A	3B - 0A	4B - 1A	5B - 2A
9	0A - 5B	1A - 0B	2A - 1B	3A - 2B	4A - 3B	5A - 4B
10	0B - 5B	1B - 0B	2B - 1B	3B - 2B	4B - 3B	5B - 4B
11	0A - 4B	1A - 5B	2A - 0B	3A - 1B	4A - 2B	5A - 3B
12	0B - 4B	1B - 5B	2B - 0B	3B - 1B	4B - 2B	5B - 3B
13	0B - 3B	1B - 4B	2B - 5B	3A - 0B	4A - 1B	5A - 2B

Table 5.6 Computation scheme in a ring communication structure

5.4 Conclusion

In this section we make a brief comparison of the five communication structures. As shown above the calculation time is the same for all structures, thus we only need to compare the efficiency and speed-up of the structures with respect to its communication time. Most of the computation time is used by the calculation itself and not for communication. Therefore communication time is less important, when more molecules must be calculated. In our examinations, we took a simulation with 1000 and one with 10,000 molecules. We compared systems with up to 64 processors.

Communication Structure	Communicatio Latency	Reason
bus	l=1	The simplest architecture. Communication net is not used during computation.
2D-net	l=3	Communication relatively simple. Every communication node has to handle two communication links and memory access.
hyper cube	l=3	This is the 3D-expansion of the 2D-net. The distances between the PEs is not larger than that on the 2D-net, which results in the same latency.
recursive structure	l_l=3, l_g=5	The local communication is as in the 2D-net, because we have 4 clusters with 4 processors. The global communication is the sum of the local communication and that of the 2D-net minus 1, then the global communication nodes can access directly via the local communication nodes.
ring	l=2	Communication is very simple, because you have only unidirectional communication links.

Table 5.7 Communication latencies

The following figures show the speed-up of the different communication systems on a simulation with 1000 and 10,000 molecules:

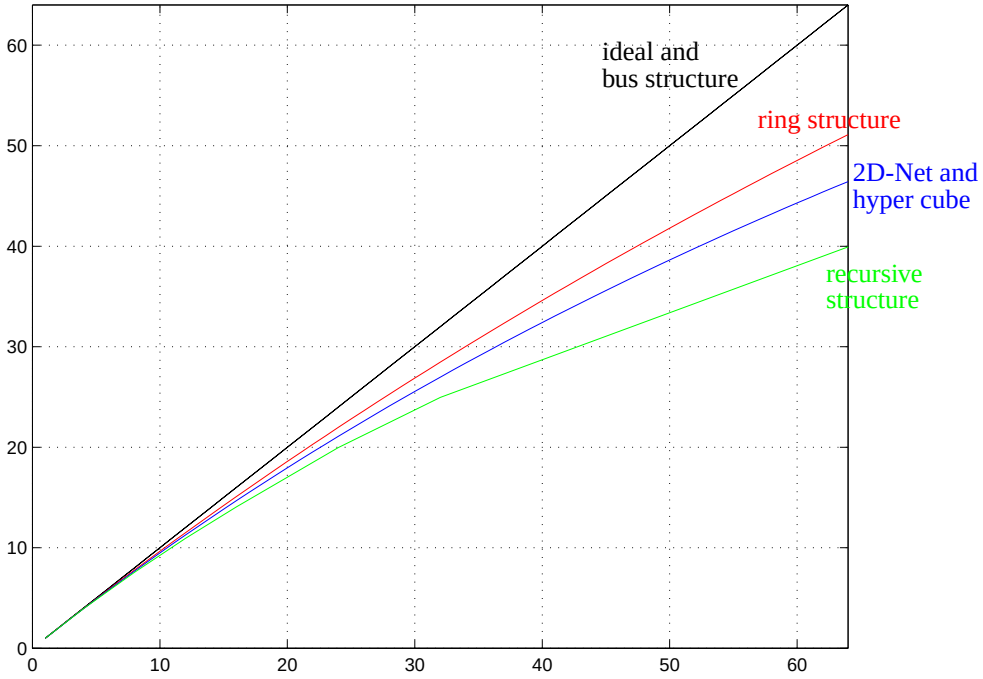


Fig 5.10 Speed-up of different communication structures with 1000 molecules

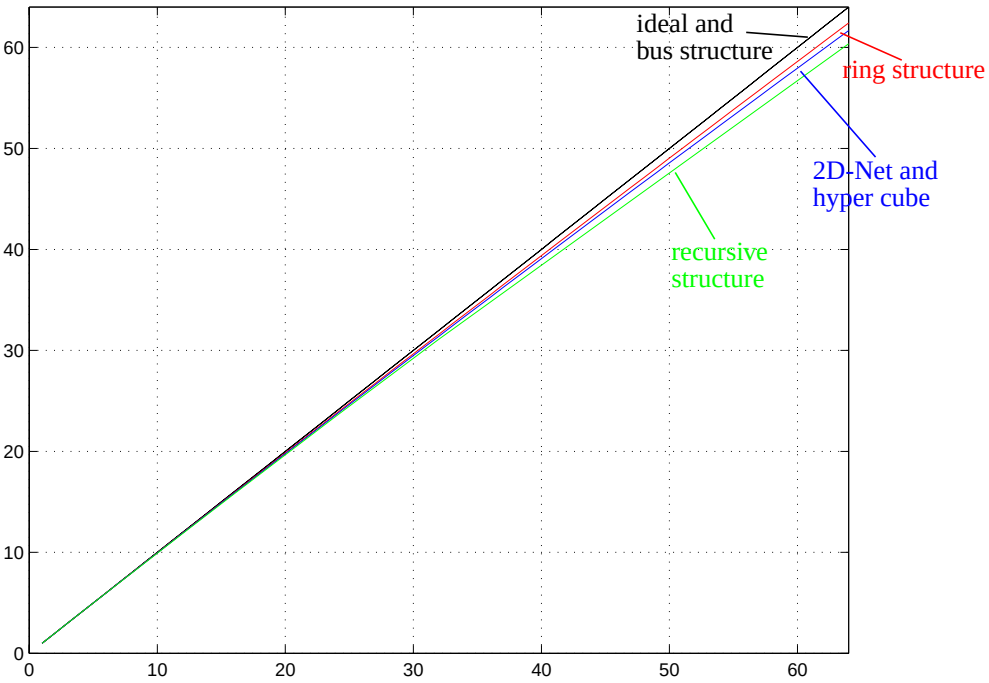


Fig 5.11 Speed-up of different communication structures with 10000 molecules

On Fig. 5.10 and Fig. 5.11 we see, that the communication loss on a ring architecture is the smallest, on a 2D-net and a hyper cube a little larger and on a recursive structure the largest. Additionally we learn that the communication loss matches less if we have more molecules.

In the next table we have listed the advantages and disadvantages of the different communication structures:

topic	**bus**	**2D-net**	**hyper-cube**	**recursive structure**	**ring**
calculation distribution	complex	simple	simple	simple	simple
data distribution / memory use	bad	good	good	good	good
computation scheme	not used	complex	very complex	complex	simple
speed-up	ideal	medium	medium	bad	good
expansion possibility	simple	difficult	difficult	medium	simple

Table 5.8 Comparison of the different communication structures

Although the bus structure has the best speed-up, hardware costs are relatively high, because all molecules are distributed onto every PE's memory. Therefore a large memory is needed. Additionally it's relatively complex to define, which processor calculates which molecule-pairs.

Therefore we see, that the ring structure is the best for the $O(N^2)$ pairlist calculation problem. It's structure is simple, few memory is used and the structure and expansion possibility are easily feasible. Further ring structure has the fewest communication loss.

6 Model Refinement

Goal: exact performance models for the workstations and communication systems for different target architecture. The results of this chapter can be used as input for the Mathematica and Codesign model described in paragraph 7. This is not done by now, the parameters actually used in paragraph 7 are these described in paragraph 3 and paragraph 5.

We made benchmarks on the ADSP-2106x SHARC DSP processor simulator to compare its performance with an FPGA and a host computer. The ADSP-2106x SHARC can be either programmed in C or in Assembler. The programming source of the GROMOS software is in Fortran. We translated first the Fortran routines into C, have then optimized the code for the Sharc DSP and at last we have translated it into Assembler, thus it is possible to use specialities of the Sharc to improve speed.

The benchmark examples used in this chapter are a simple distance calculation algorithm to compare the Sharc with an FPGA and a tightly more complicated pairlist algorithm to compare the Sharc with the host computer.

6.1 Distance Calculation with the Sharc DSP

The distance algorithm is a good part of GROMOS to compare the speed of the SHARC with the speed of a hardware implementation on an FPGA. For distance algorithm only adders, subtractors and multipliers are needed. In addition, this algorithm is the most used for calculating the nonbonded forces in GROMOS because it is used for force, energy and virial calculation.

6.1.1 Method

The distance algorithm is a single programming routine. The coordinates of two atoms are sent to the program and it gives back a distance vector and the square of the distance. Different spatial arrangements with different kinds of periodic boundary conditions can be modelled with the distance algorithm in GROMOS. First, the distance algorithm can calculate the distance of atoms, which are in the vacuum. Second distances of atoms in a space, with periodically boundary boxes, can be calculated. This boxes can also be distorted. The third is a space with truncated octahedrons. Additionally the atoms can also be in a space with four dimensions. Therefore a three- or a four-dimensional distance can be calculated with this routine. As concerned on its physical definition and chemical requirements the distances in the octagonal space and the space with the distorted boxes can only be calculated in three dimensions.

There are many conditions in the distance algorithm program. First the distance vector is calculated. Then the distance is calculated for atoms, which are in the vacuum, otherwise the distance vector is adjusted to the space with periodic boundary boxes and the distance is calculated. If we have a space with distorted boxes with periodic boundary octagons, the distance vector is adjusted to this structure.

The components of the distance vector is calculated sequential. Therefore many loops are programmed, where the components of the distance vector are calculated in three or in four dimensions. The following graph shows the structure of the program.

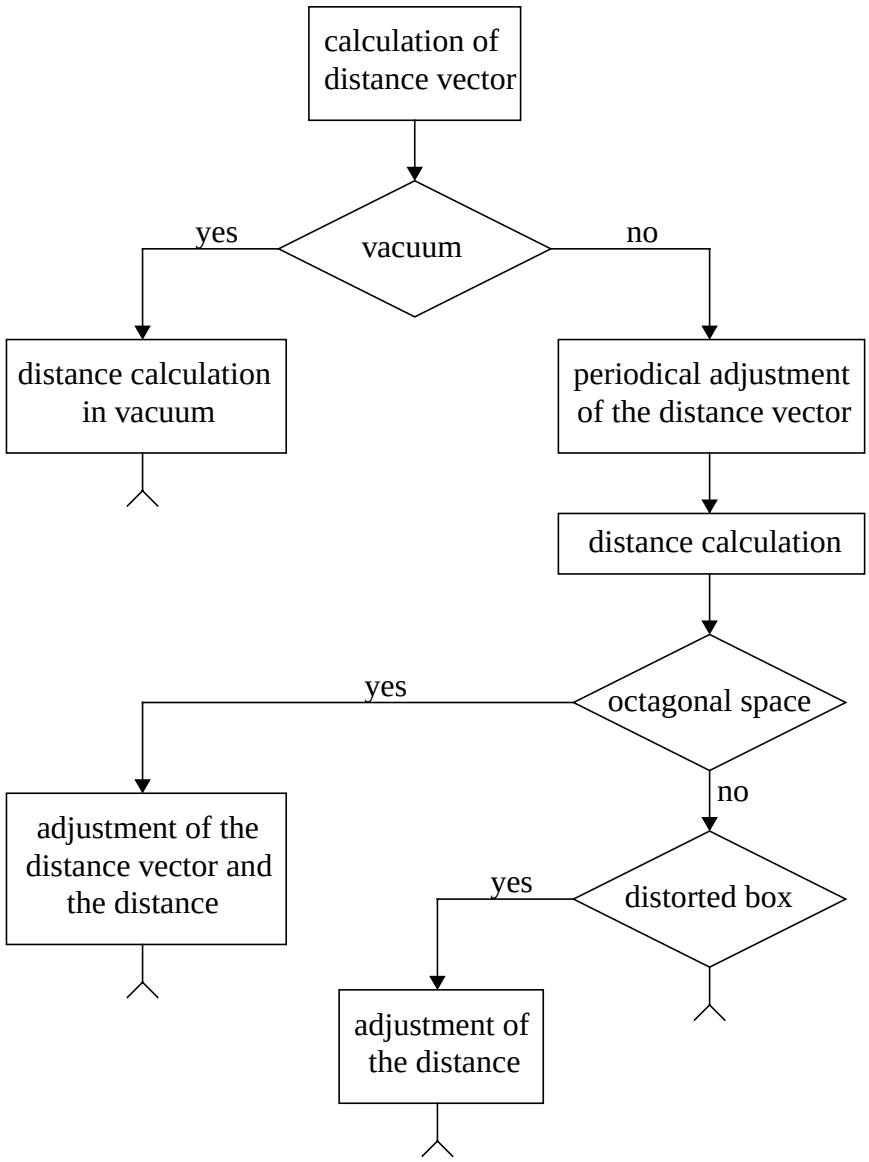


Fig 6.1 Distance algorithm

The ‘periodical adjustment of the distance vector’ runs as follows. Every component of the distance vector is first compared with the positive value of the half length of the periodic boundary box. If the component is higher, the side length of the box is subtracted. Else the component is compared with the negative value of the half length of the box. If it is lower, the side length of the box is added.

As a lot of if-statements are used in the whole program routine, additionally a lot of if-statements are used in this part of the program routine.

6.1.2 C-Program Optimization

To migrate the distance algorithm from GROMOS to SHARC, we took firstly the routine in Fortran source code and translated it directly into C. Then we tested its correctness in the SHARC DSP simulator. After that, we made the first benchmark. The SHARC compiler has an automatic optimizer for the C-code. Therefore in table 6.1 two values are listed, the first with no optimizer, the second with automatic optimizer.

For benchmarking we took the time, which is used by processor, for calculating the distance vector and the distance from only one pair of atoms. There are many possible runs through the program and we have all of them examined:

- Atoms in vacuum can be calculated in three and four dimensions.
- There are three possible runs, if distances in a space with periodic boundary boxes are calculated. The first is, that both atoms are in the middle of the box and their distance vector must not be adjusted. The second is a positive adjustment and the third a negative adjustment of the distance vector. This can also be calculated in three or four dimensions.
- The same runs are possible for the distorted box, but only in three dimensions.
- If atoms in a space with periodic boundary octagons are calculated, we have also a periodic adjustment of the distance vector first and after then an adjustment of the distance vector for the octagonal space. If both atoms are in the middle of the octagon, no adjustment is needed, this is the fourth possible run.

The possible runs and its benchmark values are listed in table 6.1. After the benchmarking of the C-routine, we made different manual optimization:

1. We made a simplification of the if-statements to make the code more efficient. We adapted the if-conditions to the internal structure of the SHARC-processor.Additionally we moved some if-statements to another place in the routine, such that fewer if-statements are executed in certain runs.
2. We replaced the loops with sequential code. Although the SHARC has a program sequencer, which controls loops more efficiently than in a normal processor, it's faster to code the loops sequential in the program, because the loops in the program are executed only for three or four times.
3. We removed all temporary variables. This caused an interesting effect, when we used the automatic optimizer.

Dim	Space	Adjustment	Optimization Step			
			Origin	1.	2.	3.
3	vacuum	no	219/159	219/163	106/55	106/55
	cubic	no	309/225	309/229	194/116	172/121
		positive	303/216	303/220	188/107	169/118
		negative	330/237	330/241	206/122	190/139
	octahedron	no	333/247	333/251	218/138	196/143
		positive	514/399	509/396	284/193	265/204
		negative	541/420	536/417	290/196	274/213
		octagon only	520/408	515/405	290/202	268/207
	monoclinic	no	330/241	330/245	215/130	193/135
		positive	324/232	324/236	209/121	190/132
		negative	351/253	351/257	227/136	211/153
4	vacuum	no	287/211	259/193	133/67	133/67
	cubic	no	404/296	372/275	244/144	218/152
		positive	396/284	364/263	236/132	214/148
		negative	432/312	397/289	260/152	242/176

Table 6.1 Benchmark of the distance algorithm in C on SHARC

In table 6.1 we have listed all benchmark values. The values are give in number of clock cycles, which are used by SHARC. If we look at the values in the 2nd and 3rd column, we see, that the values with no automatic optimization in the 3rd column are lower than that in the 2nd. But the values with automatic optimization in the 3rd column are higher than that in the 2nd column.

The reason is, because we removed all temporary variables in the 3rd optimization step. This causes a more efficient code if we do not use the automatic optimizer, because the values are not copied into temporary variables, which are stored then in memory. If we use the automatic optimizer, we got a better result, compared to using temporary variables, because the automatic optimizer recognizes temporary variables and uses registers in the register file for it. Then the calculation is done with using more registers, what makes the calculation more efficient. If we have no temporary variables defined, the optimizer uses so few registers as possible and all temporary values are stored in memory, which needs more time.

6.1.3 Assembler Optimization

The best performance on a DSP is reached with Assembler program coding. Therefore we used this program language to make further speed optimization.

By examining the compiled assembler code of the best C program version, we saw that there are mainly three properties of the SHARC to make the program faster, which was not used by the C code and must therefore be introduced manually.

We optimized the assembler code in four steps and made the following optimization:

1. The compiler uses registers of the DAG to fetch values from the memory and to store values of the register file into memory. This needs extra clock cycles for storing the pointers into the DAG registers. We changed the program so, that fetches from and to the memory are done by direct addressing.
2. In the C code, many intermediate results are stored in the internal memory, although there are registers in the processors register file, which are not used. This needs more time. Therefore we optimized the code for better using the register file.
3. The SHARC has a instruction pipeline with three steps: *fetch* - *decode* - *execute*. Therefore it can execute one instruction per clock cycle. If there is a branch, the program pointer is set to the branch destination only(erst) in the *execute* stage. Thus the pipeline is filled with the two instructions, which are placed immediately after the branch in the program code. These are then in the *fetch* and *decode* stages of the pipeline and the pipeline must be refilled with the new instructions after branch is executed. Thus the program flow idles for two clock cycles.

   There are a special instruction to avoid this and is called *delayed branch*. In the *delayed branch* the two instructions in the program code, which are placed after the branch instruction are then executed too. In this way the two instructions before the branch instruction can be placed after the latter and the branch instruction can be changed into a *delayed branch*. The C compiler does not use this instruction and we have optimized the Assembler code with this in our first optimization step.

4. On SHARC many arithmetic functions and memory transfer instructions can be parallelized. Nothing of them is done, if C code is used. In the last optimization step we tried to parallelize the instructions as lot as possible and we reached a high speed-up in comparison with the last optimization step once again.

In the following table we have listed the benchmark values of the Assembler code versions. They are separated in all possible runs of the program again. Additionally the values of the fastest C code are listed for comparison too. All values are give in number of clock cycles.

Dim	**Boundaries**	**Adjustment**	**Optimization step**				
			best C	**1.**	**2.**	**3.**	**4.**
3	vacuum	no	106/55	50	42	37	23
	rectangular	no	172/121	110	81	76	50
		positive	169/118	104	75	64	48
		negative	190/139	110	81	76	50
	octahedron	no	196/143	117	84	79	51
		positive	265/204	145	101	89	60
		negative	274/213	151	104	96	65
		octagon only	268/207	151	107	101	62
	monoclinic	no	193/135	120	86	81	54
		positive	190/132	114	80	69	52
		negative	211/153	120	86	81	54
4	vacuum	no	133/67	52	44	43	24
	rectangular	no	218/152	112	86	83	48
		positive	214/148	104	78	67	45
		negative	242/176	112	86	83	48

Table 6.2 Benchmark of the distance algorithm in Assembler on SHARC

6.1.4 Conclusions

In fig. 6.2, we made a comparison of the calculation time of every space, in which the atoms can be calculated. For the spaces, on which more possible runs in the program are possible, we took the worst case of the program execution time.

On C we considered only the runtime of the origin program, the runtime of the origin program with automatic optimization, the runtime of the best manual only optimized code and the runtime of the best C code, which is manually and automatically optimized. On Assembler, we considered the runtime of these codes, on which the optimization had the highest effect (fig. 6.2). The effect of the optimization is well seen for every space in the diagram. The calculation times are taken by a SHARC with a clock frequency of 40 MHz.

In table 6.3 the runtime of the C code, which was directly translated from Fortran and the runtime of the fastest C and Assembler codes are listed.

Fig. 6.3 shows the speed-up, which we have reached with our optimization. As in the diagram of Fig. 6.2, we compared the speed-ups of every space, in which the atoms can be calculated and drawn a separate speed-up line for each.

In fig. 6.3 we considered the speed-up of the same C and Assembler programs, of which we have taken the runtime in fig. 6.2.

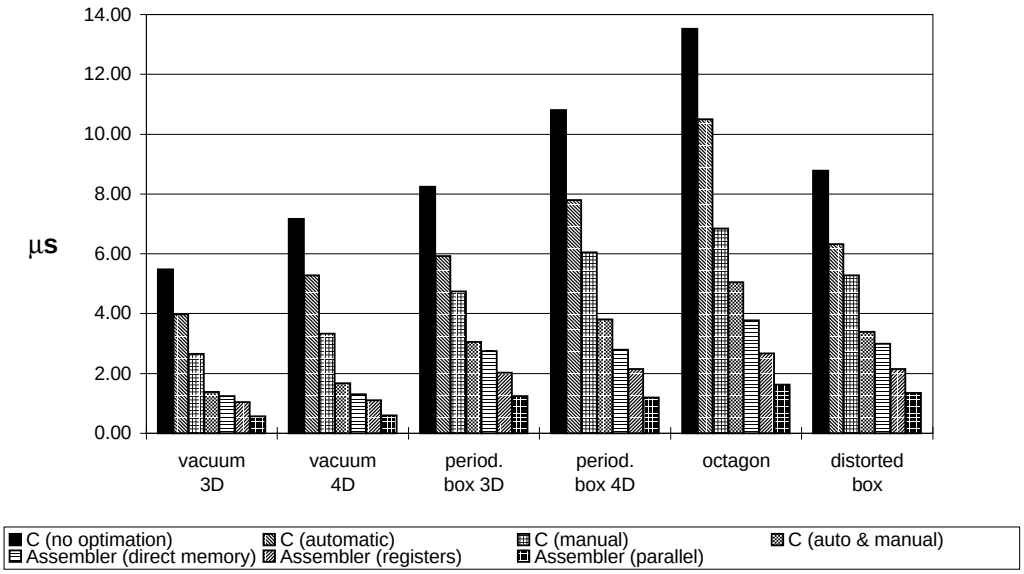


Fig 6.2 Runtime of the distance routine on SHARC

Programming Language Version	Vacuum		Rectangular		Octahedron	monocl.
	3D	4D	3D	4D	3D	3D
C (directly from Fortran)	5.48	7.18	8.25	10.8	13.53	8.78
C (fastest code)	1.38	1.68	3.05	3.8	5.05	3.4
Assembler (fastest code)	0.58	0.6	1.25	1.2	1.63	1.35

Table 6.3 Run time of the distance routine in slowest C code and fastest C and Assembler codes

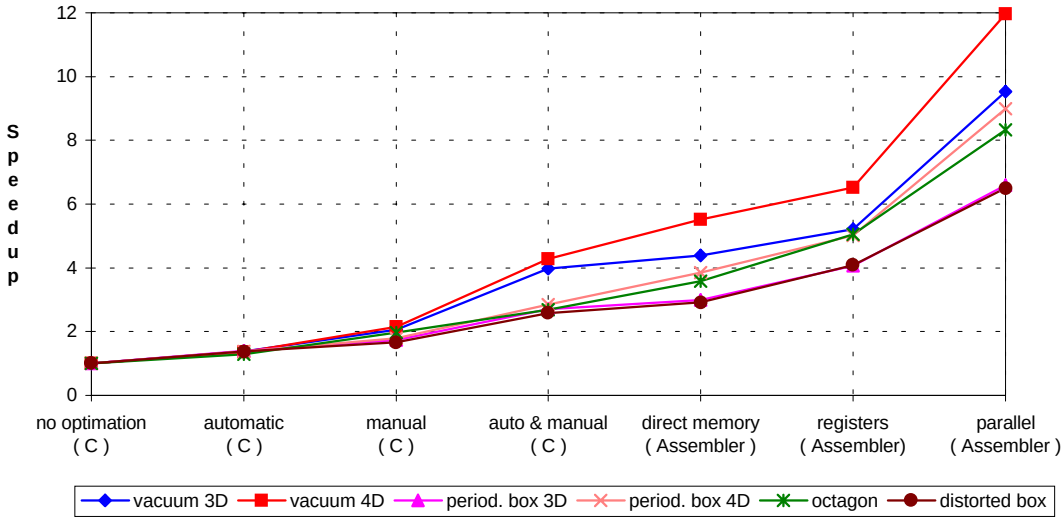


Fig 6.3 Speed-up of the distance routine on SHARC

If we look at these lines, we see first, that with manually optimized C code the optimizer is able to make a better automatic optimization. Further speed-up can be reached with register

optimizing in Assembler and at last the most speed-up is reached with parallelism of the processor operations.

The following table shows the reached speed-up in C and Assembler in contrast to the direct translated C program from Fortran and additionally the reachable speed-up of Assembler in contrast to the best C code is listed:

Programming Language Version	Vacuum		Rectangular		Octahedron	monocl.
	3D	4D	3D	4D	3D	3D
C (fastest code)	3.98	4.28	2.7	2.84	2.68	2.58
Assembler (parallel)	9.52	11.96	6.6	9	8.32	6.5
fastest C code in contrast to fastest Assembler code	2.39	2.79	2.44	3.17	3.11	2.52

Table 6.4 Speed-up of the distance routine in fastest C and Assembler Code

The effort for Assembler programming is much higher than that for C programming. We needed three times longer for the Assembler programming and code optimization than for C code optimization. But we see, that with the SHARC processor an average speed-up of **2.5** is reachable on using Assembler programming!

6.2 Distance Calculation in Hardware using FPGA's

The required accuracy for the pairlist is in the range of 0.1nm to 0.5nm. If we assume a simulation box length of 10nm an 8 bit fixed point distance calculator may fulfil the demand. The behavioural VHDL specification according Fig. 6.4was synthesized with the Synopsys behavioural compiler and implemented in a Xilinx FPGA XC4025-5.

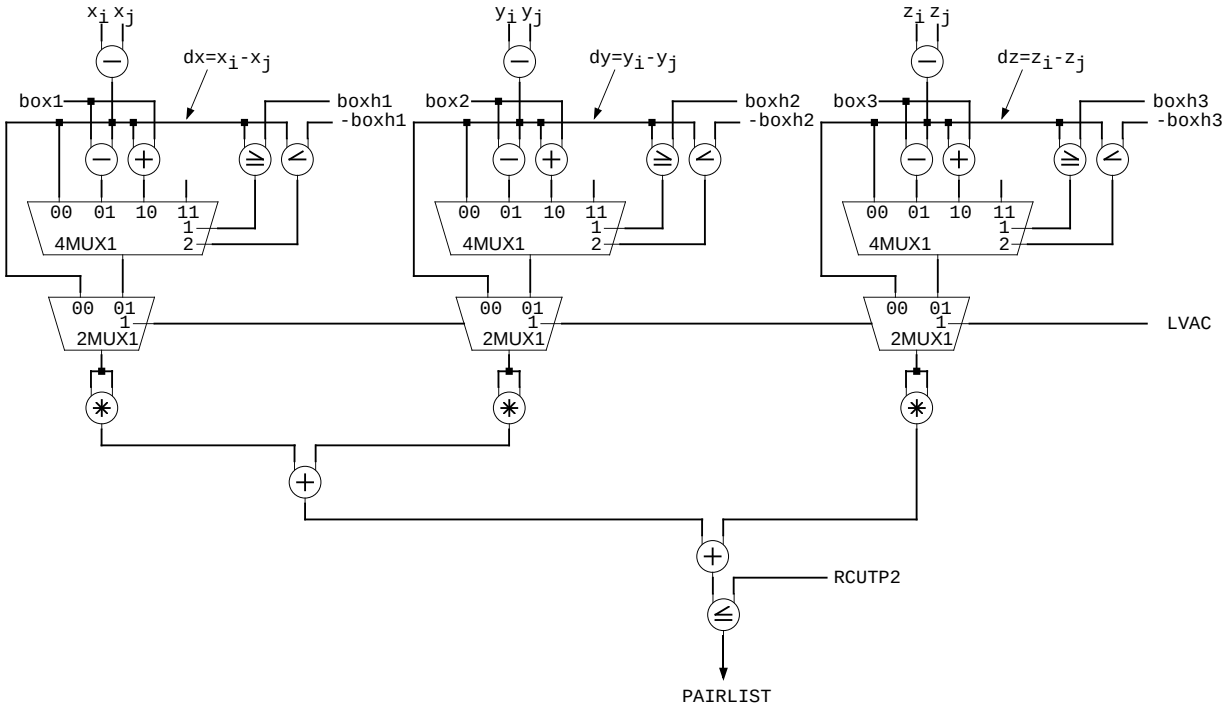


Fig 6.4 Pairlist distance calculation block diagram

After reset, the parameters (cutoff radius, box dimensions, LVAC) are read in. LVAC is a boolean specifying if periodic boundary conditions applies or not. The output is also a boolean indicating if the pair goes into the pairlist or is skipped. A two stage pipeline was introduced to increase the total throughput. With the mentioned target the maximum clock frequency is 6.4 MHz, with a valid result on the output every 2 cycles we could test 1.6 million pairs per second.

Given the Thrombin benchmark with a 1.4nm cutoff radius 20 million comparisons are needed for one pairlist construction. The sun accomplish in 9.5 seconds, the FPGA need 12.5 seconds. The relatively low performance of the FPGA implementation may be explained with shortcomings in the synopsys behavioural compiler, and a rather old fashioned and slow FPGA. We did not optimized anything by hand, so there are further possibilities for optimization.

7 Gromos MD-Algorithm Specification

In this chapter we describe two specification models and their implementation in Mathematica and Codesign [25]. The Mathematica model uses the performance and communication scheme as described in paragraph 5.1and paragraph 5.2, the Codesign description varies slightly because of compatibility restrictions of the system synthesis tool [24].

7.1 Specification in Mathematica

This model was initially developed to implement the data dependencies on the Gromos MD algorithm, e.g. the number of floating point operations depending on typical problem parameters (number of molecules, cutoff radius, etc.). The model also provides numeric values for data transfer rates and the memory requirement in dependence of the problem parameters. The further development included models of communication channels and coprocessor architectures. Within the environment it is possible to calculate the time for one MD step with or without pairlist calculation, speed-ups for different coprocessor architectures and function mappings, etc. The function mapping and the schedule are hand-coded and costly to change.

In our example we chose a coprocessor architecture as in fig. 4.18 with ten SHARC processors for the solvent-solvent force calculation. The implemented schedule forces the pairlist calculation on the dis/join processor. Fig. 7.1 shows the distribution of the coordinates according the spatial division of the simulation box. Note that for one coordinate five words must be communicated (three atom coordinate components, one atom sequence number, box member coordinates or number).

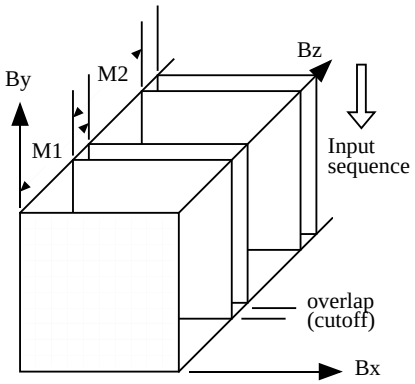


Fig 7.1 Spatial decomposition

A lot of functions must be evaluated on host, e.g. to write and read files a Fortran function must be used. This leads to the situation of maximal reachable speed-ups for as many coprocessor power you like, depending on the number of tasks forced to the host. The maximum speed-up for a Sun Ultra 1 is reached with only two force processors (Fig. 7.2). Note that in this example only solvent-solvent forces are permitted to be calculated on the coprocessor.

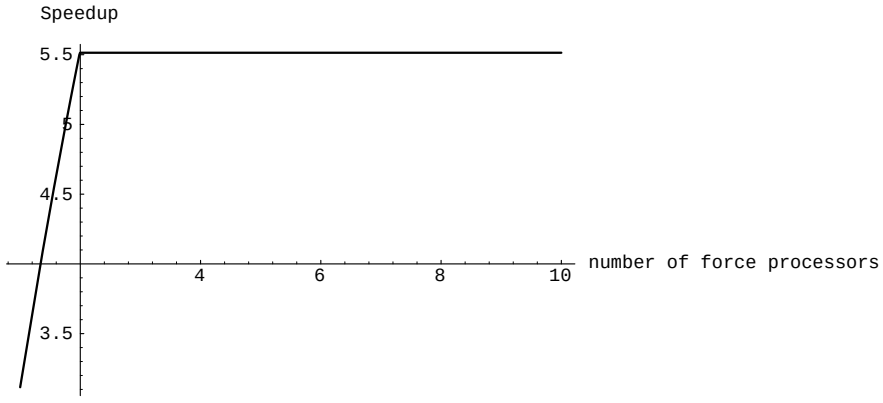


Fig 7.2 Speed-up with a 41 Mflop host

If we design a coprocessor with ten SHARC processors, the performance of the workstation must be increased by a factor of 5. So a final implementation with 10 SHARC's seems to be a well-balanced system for future workstations. Mapping all nonbonded forces and the pairlist on the coprocessor, we reach a speed-up of about 25 with the fastest workstation available in two years, accelerated with the new parallel hardware.

7.2 High level Synthesis using GP

The goal is to find an optimal system architecture using formal methods: Specification with object-oriented Petri nets, partitioning with constraints using Genetic Programming (GP) [24]. The latter task includes the optimization for minimal latency and finding the best schedule. The scheduling problem can be outlined as follows: a set of tasks has to be executed on a set of resources where each task has a certain execution time. There exist dependencies between the tasks. One looks for an assignment of the tasks to the resources that minimizes the total execution time of all tasks. Usually the dependencies are displayed in a task graph as shown in fig. 3.2. The nodes of the graph are the tasks and the dependencies are given by the edges in the graph.

In high level synthesis the nodes are viewed as operations (activities, small programs, etc.), and the edges describe the data dependencies between the operations. The resources are the available hardware units (multiplier, CPU's, etc.). A valid schedule is then given by an assignment of the starting time to each task, so that all dependencies are satisfied and not more than the available resources are required.

A scheduling algorithm is an algorithm that takes the task graph as input and returns a valid schedule as output. In the Mathematica specification scheme in paragraph 7.1 scheduling was done by hand. The integer linear programming algorithm guarantees to find the optimal solution but is unsuited as it needs exponential computation time on the input data. A lot of heuristic algorithms exist that are content with a near optimal solution at affordable computation time. Examples are ASAP-, ALAP- or list-scheduling.

Here we try to find an optimal mapping of functions to the architecture graph and finding a latency minimizing schedule for it using a GP tool [24].

7.2.1 Codesign

The *Codesign* Tool [25] is a graphic system modelling environment based on high-level object-oriented timed Petri nets. Other formalism can be embedded into the model due to the object-oriented structure. We used the tool as a graphic editor to draw the problem and architecture graph (fig. 7.3 and fig. 7.4).

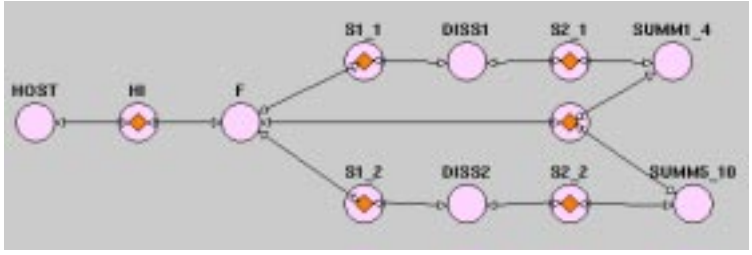


Fig 7.3 Architecture graph

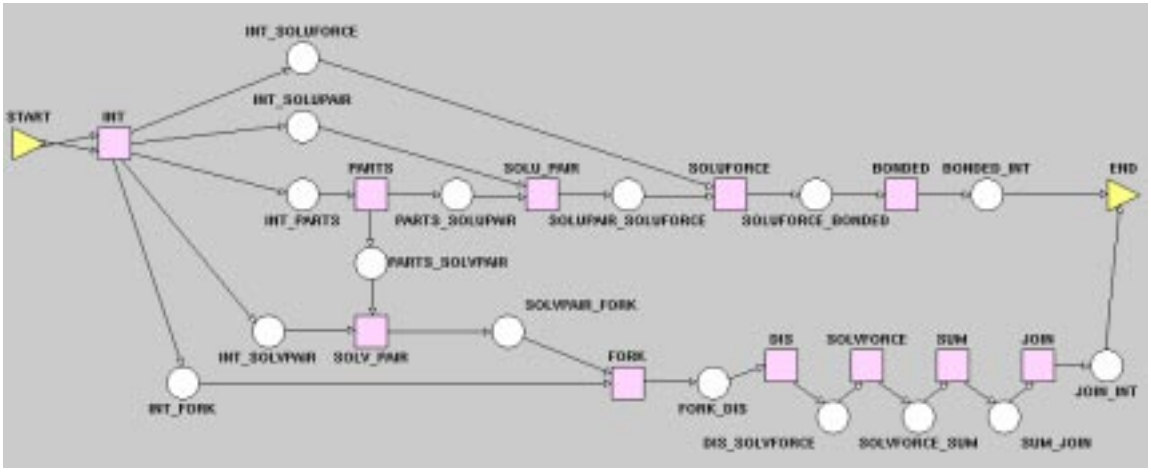


Fig 7.4 Problem graph

The architecture graph corresponds to the SHARC hierarchic approach of fig. 4.18, with one combined fork/join node, two dis/sum processors and totally ten force processors. Note that not all communication links are shown, the five force processors per dis/sum node are treated as one processors with five times the performance of one SHARC. The communication requirements are adopted accordingly. The problem graph is not simplified and reflects the real MD algorithm.

7.2.2 System Synthesis using Evolutionary Algorithms

The specification model consists of three main components:

- The algorithm to be mapped on a architecture in addition to the class of possible architectures described with a dependence graph (fig. 7.3 and fig. 7.4).
- User-defined mapping constraints between algorithms and architectures are described in a specification graph. Additional functional constraints are associated to the nodes.
- An activation is associated to nodes and edges of the specification graph characterizing allocation and binding

The specification graph is not shown due to the high complexity. Refer to the literature for examples [24]. The concept of a dependence graph is used to describe algorithms as well as architectures on different levels of abstraction. For example, the dependence graph to model dataflow dependencies on a given algorithm will be termed problem graph. The architecture including functional resources and buses can also be modelled by a dependence graph termed architecture graph consisting of hardware and communication resources.

An evolutionary algorithm solves the system synthesis problem considering a design space exploration by exploring the Pareto set of implementations. The exploration technique takes into account: (1) The communication requirements, (2) finite computation and communication resources and (3) loop pipelining (iterative schedule).

Exploration is an iterative optimization task repeating the steps a, b, c. To start an exploration, one has to specify an optimization goal, e.g. latency minimization (period) under resource constraints (cost). The input for the first iteration is a randomly generated popula-

tion of feasible bindings. The result of each step is a set of implementations, illustrated as points in fig. 7.5.

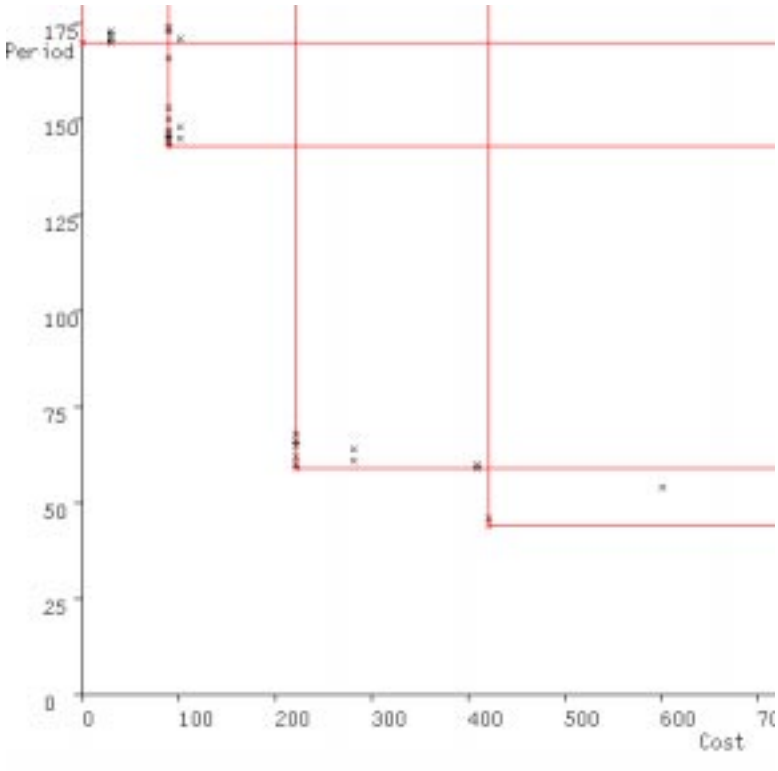


Fig 7.5 Solutions in the search space after the last iteration

The best solutions are chosen by a *fitness function* (selection). The Pareto points indicated by lines in fig. 7.5 generally achieve a high score. The initial population size then is restored through recombination (crossover or mutation) in order to exploit new points in the search space. Iteration is aborted when no better implementations are found.

For all solutions in fig. 7.5 there exist a feasible binding and a schedule such as the one in fig. 7.7. Depending on cost or time constraints, one or two of the Pareto points serve as final implementation with the appropriate mapping and schedule.

7.2.3 Design Space Exploration

We have chosen the architecture graph of fig. 7.6 and the problem graph of fig. 7.4. The parameters are assigned appropriate for one fork/join processors, two dis/sum processors and ten force processors.

Fig 7.6 Simplified architectures graph

The first three columns in table 7.1 correspond to the three “fastest“ Pareto points in fig. 7.5, where all valid implementations after 30 iterations are listed.

Host performance	Number of coprocessors	Iterations	Cost	Time
Ultra1/170=1	13	30	420	43
1	7	30	220	57
1	3	30	90	143
2	13	30	440	34

Table 7.1 Comparison

The fastest implementation is a hierarchy with one fork/join and two dis/sum processors and five Sharc’s per dis/sum processor. The second column in table 7.1 is also a three level hierarchy, but with only one dis/sum processor and thus five force processors. The third column is a solution without force processors, the forces are calculated on the two dis/sum processors. The last column is derived from another exploration run with all parameters kept the same except the host performance. The resulting fastest architecture is the same as this in the first column, the corresponding schedule is illustrated in fig. 7.7, where the step-by-step communication through the hierarchy is apparent. The cycle time is limited by the task *soluforce* which is forced to be executed on the host. To use the performance of 13 coprocessors we need either a faster host or we must allow the evolutionary algorithm to map *soluforce* to the coprocessors.

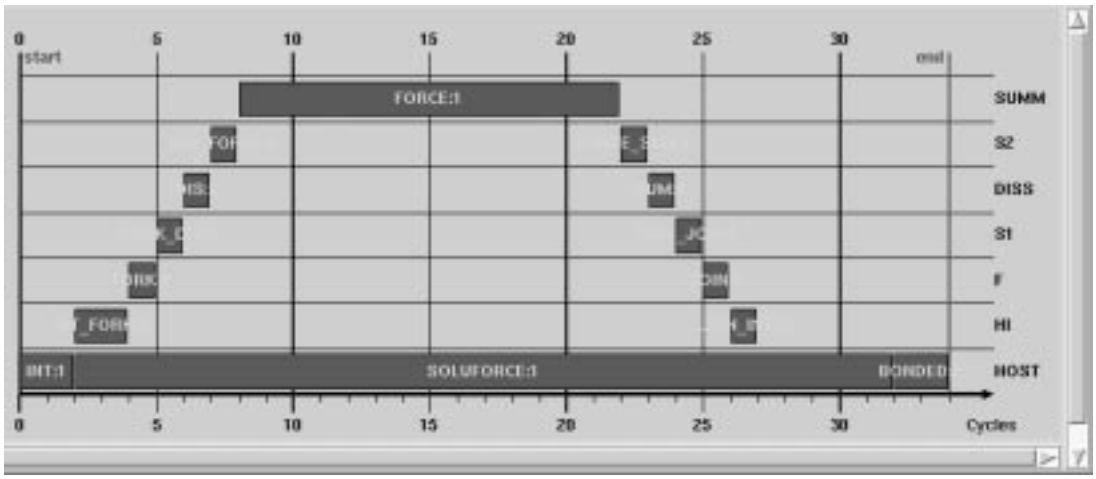


Fig 7.7 Schedule of the fastest implementation

After exploring all problem graphs the most promising solutions (hierarchical DSP, ASIC processors, workstation cluster, distributed memory RISC multiprocessor) are further investigated: A better architecture model is combined with a generic model of the MD algorithm (CDFG) and implemented in Mathematica as in the previous chapter. This task is still manual but necessary if we want to emulate a real MD step.

8 Conclusion and Further Work

The investigations presented in this report show that a pure ASIC solution as a coprocessor is not suitable to achieve our main goal. Only if the coprocessor can calculate all nonbonded forces, an acceleration factor of ten is possible. A typical MD run is a simulation of one or more identical proteins dissolved in solvent. One protein may have thousands of atoms, dissolved in ten thousands of solvent atoms.

Possible architectures are mixed solutions with ASIC's for pairlist and distance calculations, eventually for the solvent forces calculation, and standard processors (DSP, RISC) for the solute nonbonded forces. The pairlist and the nonbonded forces calculation algorithms are easy to parallelise and may be executed on the same RISC processors. Obviously a homogenous hardware architecture without custom chips is much easier to develop, implement and maintain. A mixed solution is complicated due to device driver issues, hardware complexity, time to market, and price. Therefore, the focus is now on RISC solutions interconnected with a general purpose network like Myrinet or interconnected with a new custom network optimized for low latency. The RISC processor boards have a simple structure and may consist of several processors on one board or of scalable single processor boards. Possibly such boards may be bought with an operating system and device drivers.

The further work includes more profiling on the newest processors like DEC alpha 21064 and 21164, MIPS and PowerPC. With these profiling results similar design space exploration and modelling techniques will be used to find a final implementation. Depending on this hardware structure the processing elements and/or the communication system must be developed, built and tested. With the modelling technique and the design space exploration methods presented in this report, it is rather easy and straightforward to check new architectures and solutions against our requirements. Depending on the interconnection network, existing or new PCI device drivers for IRIX and Solaris must be customised for the coprocessor. Target machines are SUN Ultra 30 and Silicon Graphics Octane workstations.

In addition, the Gromos software must be adapted for parallel coprocessors. A C code version of a new pairlist algorithm as well as the nonbonded forces routines are parallelised and tested. The entry point of the new software is well defined and established in co-operation with the Computational Chemistry Group. Special care must be taken because Fortran code and C code coexist. A detailed software concept is under development. Generic parallelisation should be possible for different hardware structures or acceleration techniques. For this purpose, one can imagine a new specification language for the target hardware, the software partitioning and the algorithm specification. With that, automatic generation of code is possible.

Literature

Related Hardware Projects

[1] A.F. Bakker, C. Bruin: Design and Implementation of the Delft molecular-dynamics processor. Special purpose computers, 183-222, Academic Press Inc. (1988)

[2] W. Scott, A. Gunzinger: Parallel molecular dynamics on a multi signal processor system. Computer Physics Communication 75, 65-86, (1993)

[3] T. Fukushige, J. Makino: WINE-1: Special-purpose computer for N-body simulations with a periodic boundary condition. Publ. Astron. Soc. Japan 45, 361-375, (1993)

[4] T. Ebisuzaki, T. Fukushige, J. Makino: GRAPE Project: An Overview. Publ. Astron. Soc. Japan 45, 269-278, (1993)

[5] T. Ito, J. Makino, T. Fukushige: A special-purpose computer for n-body simulations: GRAPE-2A. Publ. Astron. Soc. Japan 45, 339-347, (1993)

[6] J. Makino, M. Taiji: (directly from makino@chianti.c.u-tokyo.ac.jp)

GRAPE-4: A massively-parallel special-purpose computer for collisional N-body simulations.

GRAPE-4: A special-purpose computer for gravitational N-body problems.

GRAPE-4: A one-Tflops special-purpose computer for astrophysical N-body Problem.

Astrophysical N-body simulations on GRAPE-4 special-purpose computer.

Next generation GRAPE systems.

[7] T. Fukushige, M. Taiji: A highly-parallelized special-purpose computer for many-body simulations with an arbitrary ventral force: MD-GRAPE. The Astrophysical Journal, 468: 51-61, (1996)

[8] J. Makino: Design trade-off in Grape systems. From makino@chianti.c.u-tokyo.ac.jp

[9] H.J.C. Berendsen, D. van der Spoel, R. van Drunen: GROMACS: A Message Passing Parallel MD Implementation. Computer Physics Communication 91, 43-56, (1995)

Papers

[10] W.F. van Gunsteren, H.J.C Berendson: On searching neighbours in computer simulation of macromolecular systems. Journal of Comuputational Chemistry, Vol. 5, No. 3, 272-279, (1983)

[11] R. Everaers, K. Kremer: A fast grid search algorithm for molecular dynamics simulations with short-range interactions. Computer Physics Communications 81 (1994)

[12] W. Smith: Molecular dynamics on hypercube parallel computers. Computer Physics Communications 62 (1991)

[13] G.S.Grest, B. Dünweg, K. Kremer: Vectorized link cell fortran code for molecular dynamics simulations for a large number of particles. Computer Physics Communications 55 (1989)

[14] D.C. Rapaport: Multi-million particle molecular dynamics I-III. Computer Physics Communications 62 (1991), 76 (1993)

[15] V.E.Taylor, R.L. Stevens, K.E.Arnold: Parallel molecular dynamics: Communication requirements for massively parallel machines. IEEE (1995)

[16] W. Scott: MD Profiling. IGC internal paper (June 19, 1995)

[17] M. Eisenring, J.Teich: Rapid Prototyping of Dataflow programs on hardware/software architectures. Swiss Federal Institute of Technology (ETH) Zürich, TIK, (1997)

Books

[18] M.P. Allen, D.J. Tildesley: Computer Simulation of Liquids. Oxford University Press 1987

[19] R.W. Hockney, J.W. Eastwood: Computer Simulation using Particles. IOP Publishing Ltd. 1988

[20] R. Haberlandt, S. Fritzsche, G. Peinel, K. Heinzinger: Molekulardynamik, Grundlagen und Anwendungen. Vieweg Lehrbuch Physik 1995

[21] H.P. Lenhof: Distanz- und Suchprobleme in der algorithmischen Geometrie und Anwendungen in der Bioinformatik. Dissertation der Technischen Fakultät der Universität des Saarlandes, Saarbrücken 1993

[22] T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen. Mannheim; Wien; Zürich: BI-Wissenschaftsverlag 1990

[23] W.F. van Gunsteren: Biomolecular Simulation: The GROMOS96 Manual and User Guide. Hochschulverlag vdf AG an der ETH Zürich, 1996

[24] T. Blickle: Theory of Evolutionary Algorithms and Application to System Synthesis. Hochschulverlag vdf AG an der ETH Zürich, 1997

[25] R. Esser: An Object Oriented Petri Net Approach to Embedded System Design. Hochschulverlag vdf AG an der ETH Zürich, 1997

[26] D.C. Rapaport: The Art of Molecular Dynamics Simulation. Cambridge University Press 1995

[27] D.D. Gajski, F. Vahid, S. Narayan, J. Gong: Specification and design of embedded systems. Prentice Hall, 1994

[28] Analog Devices: ADSP-2106x SHARC User's Manual