Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 33

Martin Gerber

Parallelising Molecular Dynamics for Message Passing Systems

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diss. ETH Nr. 13437

Parallelising Molecular Dynamics for Message Passing Systems

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Martin Gerber
Dipl. El.-Ing. ETH

born 15th December 1969
citizen of Langnau i.E. BE

accepted by recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Wilfred F. van Gunsteren, co–examiner

Examination date: November 26, 1999

Contents

Kurzfassung v

Abstract vii

CHAPTER 1

Introduction 1

1.1 Thesis Overview 1
1.2 Physics of Atomic Many–Body Systems 2
1.3 Molecular Systems: Models and Algorithms 3
1.3.1 Classical Equations of Motion 3
1.3.2 Potential Function 4
1.3.3 Long–Range Interaction 6
1.3.4 Periodic Boundary Conditions 7
1.3.5 QMD Methods 8
1.4 The GROMOS96 Implementation 8
1.4.1 Treating Boundaries 9
1.4.2 The MD Algorithm 9
1.4.3 The Force Field 12
1.5 Parallel GROMOS: Motivation and Main Goals 13
1.5.1 Related Tools 13
1.5.2 GROMOS96P Design Goal 14

CHAPTER 2

Methodology 17

2.1 System–Synthesis 17
2.1.1 Levels of Abstraction 17
2.1.2 System Synthesis: Definition and Motivation 19
2.1.3 The Basic Synthesis Algorithm 20
2.1.4 System Synthesis and Design Space Exploration 20
2.2 Specification and Modelling 21
2.2.1 Overview 22
2.2.2 Parallel Performance Metrics 23

2.2.3 Modelling Parallel Architectures 25
2.2.4 Communication Model 26
2.2.5 Modelling Local Performance 29
2.2.6 Algorithm Specification 30
2.2.7 Modelling of Functions 33
2.2.8 Modelling the Algorithm–Architecture Relation 34
2.3 Design Space Exploration 37
2.3.1 Modelling for System Synthesis 38
2.3.2 An Evolutionary Algorithm for System Synthesis 39
2.3.3 Design Decision 41
2.4 Software Synthesis 42
2.4.1 Overview 42
2.4.2 Sequential Programming Environment 43
2.4.3 Proposed Parallel Programming Environment 45
2.4.4 Code Generation from Control/Data Flow Graphs 46

CHAPTER 3

Design Space Exploration **49**
3.1 Existing Approaches 49
3.1.1 Specialised Hardware for Molecular Dynamics 50
3.1.2 Parallel Software Implementations 51
3.2 GROMOS96 Analysis 52
3.2.1 Benchmark Specification 53
3.2.2 Profiling 53
3.2.3 Reachable Speed-ups 55
3.2.4 Function Models 56
3.3 Modelling GROMOS for Design Space Exploration 61
3.3.1 Parallel Model Extension 61
3.3.2 Scaling the GROMOS Models 63
3.3.3 Models of the MD Algorithm 66
3.3.4 Architecture Models 67
3.4 Exploring the Design Space 71
3.4.1 System Synthesis 71
3.4.2 Performance Estimations 75
3.4.3 Assessment and Design Decision 81
3.5 Conclusions 83

CHAPTER 4

The Parallel GROMOS Algorithm **85**
4.1 Neighbour searching 85
4.1.1 The Link Cell Method 86
4.1.2 Grid Cell vs. Link Cell 88
4.1.3 GROMOS96P Pairlist Algorithm 90
4.2 Decomposition Methods 95
4.3 GROMOS96P Spatial Decomposition 97
4.3.1 Spatial Decomposition with a Rectangular Box 97
4.3.2 Domain Decomposition Algorithm 98

4.3.3 Replicated Data 101
4.3.4 Load Balancing 102
4.4 Parallel Twin–Range 104
4.5 Summary 105
4.5.1 Key Results 105
4.5.2 Decomposition Parameter Glossary 106

CHAPTER 5

Specification Model and Code Generation 107
5.1 Motivation 107
5.2 Parallel Software Concept 109
5.2.1 Architecture and Function Mapping 109
5.2.2 Parameterized Functions 111
5.2.3 Control/Data Flow Graph 112
5.2.4 Memory Management 113
5.3 Language Specification 114
5.3.1 GROMOS96P Specification 116
5.4 Automated Processing 117
5.4.1 Specification Parser 117
5.4.2 Checking the Static Semantics 117
5.4.3 Processing the Interface Declaration of Functional Nodes 118
5.4.4 Code Generation 119
5.5 Summary 119

CHAPTER 6

Performance Comparison 121
6.1 GROMOS Implementations 121
6.1.1 Hand–Coded Asynchronous MPI Implementation 121
6.1.2 Generated Synchronous MPI Implementation 122
6.1.3 Sequential Versions 122
6.2 Performance Metrics 122
6.3 Results 124
6.3.1 Test Environment 124
6.3.2 Comparison of the Execution Time 124
6.3.3 Exploring the Scalability on the SP-2 125
6.3.4 Workstation Cluster 128

CHAPTER 7

Conclusions 129
7.1 Fundamental Results 129
7.2 Future Perspectives 130

Appendices 131
A Benchmark Description 131
B Platforms and Compilation 133
B.1 Reference machines 133
B.2 Compiling Parallel GROMOS 134

C EBNF of the Specification Language 135
D GROMOS96P Specification 137
D.1 Control Flow Graph 137
D.2 Control/Data Flow Specification 138

Bibliography **143**

Curriculum Vitae **149**

Kurzfassung

Computersimulationen molekularer Systeme haben sich seit ihrer Einführung in den frühen Siebzigerjahren stürmisch entwickelt und wurden vielfältig verwendet. Noch heute ist die Simulation von molekularen Systemen ein Gebiet von wachsender Bedeutung nicht zuletzt deshalb, weil immer leistungsfähigere Computer verfügbar werden. Die Computersimulationen geben detaillierte Einblicke ins molekulare Geschehen und spielen eine Doppelrolle insofern, als sie durch numerische Berechnungen physikalischer Modelle einerseits zur Ergänzung und Ueberprüfung der analytischen Theorie dienen und andererseits durch die Möglichkeit der Variation von Systemeigenschaften den Bereich der Experimente zu erweitern helfen.

Die Methode der Molekulardynamik (*Molecular Dynamics, MD*) beschreibt die Bewegung der betrachteten Systeme durch Lösung mechanischer Bewegungsgleichungen und Berechnung der gewünschten strukturellen und thermodynamischen Eigenschaften durch Auswertung geeigneter Mittelwerte.

Die Aufteilung eines mit hohem Berechnungsaufwand verbundenen Problems auf mehrere Prozessoren ist ein Ansatz es zu lösen. Auf dem durch die Fliesskommaleistung beschränkten Gebiet der Molekulardynamik erlaubt die gleichzeitige Benutzung von mehreren Prozessoren die Grenze der möglichen Systemgrösse und/oder Simulationsdauer nach oben zu verschieben. Die effiziente Ausnutzung von Parallelarchitekturen ist jedoch mit Problemen behaftet.

Die vorliegende Arbeit hat zum Gegenstand, bestehende parallele MD Simulatoren zu vergleichen, die angewendeten grundlegenden Verfahren auf ihre allgemeine Anwendbarkeit zu untersuchen, gegebenenfalls anzupassen und neue Verfahren zu entwickeln. Für eine bestehende Simulationssoftware (GROMOS™) wird mit neuesten Methoden systematisch die optimale parallele Zielplattform und Parallelisierung bestimmt und realisiert. Im einzelnen werden folgende Themen behandelt:

1. Eine umfangreiche Vergleichsstudie der meistbekannten klassischen—das heisst nichtquantenmechanischen—MD Simulatoren wird vorgestellt. Anhand dieser Studie wird untersucht:
   - Welche Parallelisierungsarten gibt es, und wie steht die Wahl einer spezifischen Parallelisierungsstrategie im Zusammenhang mit dem eingesetzten Algorithmus.
   - Die Effizienz potentieller Zielarchitekturen wird beurteilt bezüglich der rechnerischen Leistung und den Entwicklungskosten von Hardware und Software.

- Die spezifischen Vor- und Nachteile einzelner Verfahren bezüglich GROMOS werden detailliert ausgearbeitet.

2. Zur Bestimmung der optimalen Zielarchitektur kommen folgende Methoden zum Einsatz:
   - Ein Systemsynthese-Werkzeug basierend auf einem Evolutionären Algorithmus wird vorgestellt und zur Exploration des Entwurfsraumes eingesetzt. Es wird sich herausstellen, dass diese Synthesemethode nicht geeignet ist, um für die Klasse der datenparallelen Algorithmen geeignete Implementationen zu finden.
   - Kosten und Performanz von möglichen parallelen Architekturen werden abgeschätzt anhand analytischer Modelle von Algorithmus und Architekturen. Dabei wird sich zeigen, dass gegenwärtige Modellierungstechniken paralleler Architekturen nicht geeignet sind, um beispielsweise *message-passing-basierte* Systeme zu beschreiben; im folgenden wird deshalb ein Modell zur formalen Beschreibung paralleler Architekturen auf höheren Abstraktionsebenen entwickelt.

3. Mit dem Ziel, die Programmierung datenparalleler Algorithmen signifikant zu vereinfachen, wird eine Spezifikationssprache basierend auf einem neuartigen Graphenmodell definiert. Dieser Graph repräsentiert typischerweise den datenparallelen Teil eines Algorithmus als dedizierten Kontroll-/Datenflussgraph. Der kritischste Teil eines parallelen Algorithmus ist erfahrungsgemäss die Implementierung der Kommunikation zwischen einzelnen Rechenknoten, im vorliegenden Fall anhand standardisierten *message-passing* Bibliotheken. Ein Codegenerator erlaubt es, diese Teile automatisch aus dem Kontroll-/Datenflusssgraph zu erzeugen.

Abstract

Computer simulations of molecular systems have shown a tempestuous development since they began in the early seventies and were put to widespread use. Today computer simulation of molecular systems is still an area of growing importance, due in part to significant computer performance improvements. Computer simulations allow thorough investigation of molecular processes and play a dual role in helping to improve and verify the analytical theory through numerical calculation of physical models on the one hand and helping to widen the scope of experiments by enabling the variation of system properties on the other hand.

Molecular Dynamics (MD) methods describe the dynamic behaviour of a system by solving mechanical equations of motion and calculating desired structural and thermodynamical properties through evaluation of appropriate averages.

Bringing the power of multiple processors to bear on a computationally difficult problem is one approach to solving an otherwise intractable task. In the field of Molecular Dynamics, where applications are limited by the floating point performance of computers, the computing power of multiple processors can be harnessed to increase the size of a molecular system and/or timescale available to the simulation. However, the efficient use of multiple processors poses some challenging problems.

The present thesis is apt to compare the various commonly applied parallel MD simulation tools, to investigate, and, where necessary, to adjust the presently available basic methods in regards to their general usefulness, and to develop new techniques. For one such MD simulator (GROMOS™) the optimum parallel target architecture and parallel implementation is to be found using modern methods systematically. In particular, the major contributions are:

1. A comparative study incorporating the most common classical, i.e. not quantum mechanical, Molecular Dynamics tools is presented. Based on this study, the following topics are investigated:
   - A survey of decomposition methods is provided and the choice of such a method is related to the employed algorithm.
   - The efficiency of possible target architectures is assessed taking into consideration performance as well as the development cost of hardware and software.
   - The specific advantages and drawbacks of certain methods are discussed with respect to GROMOS.

2. The following methods are used in order to determine the optimal target architecture:
   - A system synthesis tool based on an Evolutionary Algorithm is presented and employed to explore the design space in order to find suitable implementations. It will be shown that the presented synthesis method is inadequate in finding implementations for data parallel algorithms.
   - Based on analytical models of both algorithms and architectures, the cost and performance of possible architectures are estimated. It will be shown that existing parallel computing models are not suited to describe, for example, message passing systems. Therefore, a customised formal model is introduced with which parallel architectures are easily described on a higher level of abstraction.
3. Significant simplification of programming data parallel algorithms was the main goal in developing a domain specific language in order to specify the algorithm using a new graph model. A model-compliant specification represents only the data parallel part of the algorithm as a dedicated control/data flow graph. The most critical part of a parallel algorithm is the handling of message passing based interprocessor communication. With the appropriate code generator it is possible to generate these parts automatically from the control/data flow graph.

CHAPTER 1

Introduction

The simplicity of Molecular Dynamics simulation lies in its ability to evaluate the development of multi-particle systems by numerical integration of Newton's equation of motion. But, as with most simple principles, additional refinement is needed to correct relevant errors and introduce concepts and techniques to impart more practical value to an otherwise sluggish software.

Therefore, since the inception of the first computer simulation of molecules in the fifties, much work has been done resulting in the creation of different Molecular Dynamics simulation tools, and over the years parallel versions of most simulation programs have been made available.

This thesis deals with the parallelisation of the GROMOS™ (GROningen MOlecular Simulation Software) program package. Novel algorithms are designed to realise the best possible target system for our application. As an example, a specification language as well as a code generator has been developed allowing parts of the parallel code to be generated automatically.

1.1 Thesis Overview

In this chapter, an overview of the theory of atomic physics is given to help the reader comprehend the main principles of the different Molecular Dynamics simulation techniques. The features of some typical Molecular Dynamics software packages are compared with those of GROMOS, which point out several drawbacks and weaknesses of the current GROMOS distribution. This leads us directly to the formulation of a list of requirements for the parallel GROMOS version. Including some technical details in regard to the algorithms and force field used in GROMOS turned out to be essential for understanding the parallelisation strategy, which is developed in subsequent chapters.

Chapter two would serve as a reference to guide throughout the thesis. The employed methodology is introduced and explains all development stages from the list of requirements to the final parallel application. Finer details about the various steps are presented in the subsequent chapters. In particular, chapter three describes how the best suitable target architecture is found, chapter four illustrates the applied parallelisation method and chapter five introduces the specification language and the corresponding code generator. Moreover, the performance of the new parallel GROMOS versions—hand-coded and generated—is evaluated and compared with the initial sequential program in chapter six. Finally, the thesis ends with a conclusion and outlook. Details regarding the implementation are described in the appendix.

1.2 Physics of Atomic Many–Body Systems

A many-body system consisting of N indistinguishable particles, also called an N-body system, has $f = 3N - r$ degrees of freedom in the three dimensional space, where the number of cartesian coordinates is kept constant. The number r is derived from the number and kind of constraints applied to a molecular system. Constraints usually concern covalent bonds, bond angles, and even velocities or forces. The remaining f degrees of freedom are treated explicitly by the simulation tool, or implicitly by adding correction terms to the interaction function of explicitly treated degrees of freedom.

Some degrees of freedom or interactions in molecular systems are of type $h\upsilon \gg k_B T$. This means that at a certain absolute temperature T the vibrational frequency υ of these bonds is so high that exact results may only be obtained by solving the quantum mechanical equation of motion, the time dependent Schrödinger equation

$$\hat{H} \cdot \Psi(\mathbf{r}_i, t) = -i \cdot \frac{h}{2\pi} \cdot \frac{\partial}{\partial t} \Psi(\mathbf{r}_i, t) \tag{1.1}$$

with Ψ the wave function, $\hat{H}$ the Hamiltonian operator, h the Planck's constant, and k_B the Boltzmann constant. Equation (1.1) applies to all particles $i = (1, \ldots, N)$ with position vectors $\mathbf{r} = (x, y, z)$ in a system containing N particles (vectors: bold). The Hamilton function (classical) or the Hamiltonian operator (quantum mechanical) represents the total energy of the system. The usage of *operators* is typical of quantum mechanics. In contrast to classical physics where physical quantities are expressed with scalars and vectors, operators apply in quantum mechanics. As an example, the classical momentum of a point mass m moving with the velocity $\mathbf{v}$ is $\mathbf{p} = m\mathbf{v}$. The corresponding quantum momentum is given by

$$\hat{\mathbf{p}} = \frac{ih}{2\pi} \cdot \nabla(\mathbf{r}) = \frac{ih}{2\pi} \cdot \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \tag{1.2}$$

The sign '^' is used to avoid confusion between classical and quantum representatives. The definition of a quantity in classical mechanics is always also valid in quantum mechanics if it does not contain derivations (correspondence principle [20]). Therefore, $\hat{\mathbf{r}}$ is exactly the same as $\mathbf{r}$.

If the system contains only one particle at position $\mathbf{r}$, the solution to the Schrödinger equation is a wave function of the form

$$\Psi(\mathbf{r}, t) = a e^{(i\mathbf{k}\mathbf{r} - i\omega t)} \tag{1.3}$$

with $\mathbf{k}$ the wave vector and $h\mathbf{k} = 2\pi\mathbf{p}$, $\mathbf{p}$ the momentum and ω the angular frequency with $\omega = 2\pi\upsilon$. Note that in many-body systems equation (1.3) is not a suitable solution.

In quantum theory, it is not possible to calculate the exact position of a particle, but only a probability density. The probability w that a particle resides in a certain volume *dV* is

$$w = |\Psi(\mathbf{r},t)|^2 dV . \tag{1.4}$$

In the following passage, the vectors $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ and $\mathbf{p} = (\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_N)$ will be understood as an abbreviation for the position and momentum vectors of all particles in a many-body system. Then, the general wave function $\Psi(\mathbf{r}, t)$ contains all information about the behaviour of a multi-atomic system. The Hamiltonian operator $\hat{H}$ may be separated into a kinetic energy term $\hat{K}$ and a potential energy term $\hat{V}$ as follows:

$$\hat{H}(\mathbf{r},\hat{\mathbf{p}}) = \hat{V}(\mathbf{r}) + \hat{K}(\hat{\mathbf{p}}) \tag{1.5}$$

For all particles i with mass m_i and position vectors $\mathbf{r}_i$ the kinetic energy reads

$$\hat{K} = -\sum_i^N \frac{h^2}{8\pi^2 m_i} \cdot \frac{\partial^2}{\partial \mathbf{r}_i^2} = -\sum_i^N \frac{h^2}{8\pi^2 m_i} \Delta(\mathbf{r}_i) = \sum_i^N \frac{\hat{\mathbf{p}}_i^2}{2m_i} \tag{1.6}$$

Partial derivations are often described with operators, e.g. the Laplace operator $\Delta(\mathbf{r}) = \nabla(\mathbf{r})^2$ is applied on position vectors. Due to the fact that the potential $\hat{V}$ depends only on the position vectors, $\hat{V}$ is not necessarily an operator; thus $\hat{V}(\mathbf{r}) = V(\mathbf{r})$. The potential energy V determines the interaction between particles.

Alternatively to the wave function $\Psi(\mathbf{r}, t)$, each point in the *phase-space* defines one possible state of the molecular system. The phase-space is the 2N–dimensional space which is spanned by $\mathbf{r}$ and $\mathbf{p}$. If $f < 3N$ in the three dimensional space, the phase-space would not be orthogonal. To retain orthogonality, the coordinates and momenta may be transformed to so-called generalized quantities.

Numerical approximation or analytical solution of (1.1) is called *Quantum Molecular Dynamics (QMD)*. Unfortunately, it is not possible to find analytical solutions for systems of practical interest but only for the most simple systems with very few particles. Even numerical approximation of $\Psi(\mathbf{r}, t)$ will be too computation intensive for most molecular simulations. Usually it is not required to solve the Schrödinger equation since simplifying techniques allow accurate simulations of most molecular systems: depending on the desired accuracy, the types of the involved particles, and the simulation goal (energy minimisation, simulated annealing, etc.), a number of methods and algorithms may be applied to reduce calculation cost.

1.3 Molecular Systems: Models and Algorithms

1.3.1 Classical Equations of Motion

The particles in question may be separated into two groups, the protons in the atomic nucleus and the electrons orbiting the nucleus. The nuclei are treated like single particles which must not be modelled with quantum physics due to their relatively large weight and slow motion. In contrast, the light electrons move very fast and may not be modelled with classical mechanics in a first approach. Within the Born-Oppenheimer approximation [12] it is possible to express the Hamiltonian of a system as a function of the nuclear variables, the motion of the electrons having been averaged out. The result is a classical (which means not quantum mechanical) atomic model with moving point masses. Under these assumptions, the potential energy may be given in closed form (section 1.4.3, [26]). Resulting classical equations of motion can be formulated in different ways, according to Newton, Langevin [1] or Lagrange [11]. Newton's equations of motion have the form

$$\mathbf{f}_i(t) = -\frac{\partial}{\partial \mathbf{r}_i} V(\mathbf{r}) = m_i \cdot \frac{d^2}{dt^2}(\mathbf{r}_i(t)) \tag{1.7}$$

Newton's equations of motion are valid only with cartesian coordinates, where $\mathbf{f}_i$ denotes the sum of all partial forces exerted on a particle i at time t. To ensure an unified terminology, numerical integration of (1.7) is called *Molecular Dynamics (MD)*. If constraints are applied, the system simplifies because of the reduction of the number of degrees of freedom. The average effect of this simplification is expressed with an additional external force per particle. If the external force is modelled in the same way like the explicitly simulated particles, it is possible to include it in (1.7) by changing some parameters. Considering time fluctuations of the external force or friction effects of electrons at atomic level, a higher-order approximation is required: an additional force is explicitly added to (1.7), leading to the stochastic Langevin equation of motion [1]. Its numerical integration is called *Stochastic Dynamics (SD)*. A major application of SD is to approximate the effect of explicit solvent molecules on the dynamics of macromolecules: the macromolecule is simulated in vacuum with SD without periodic boundary conditions (section 1.3.4), thereby resulting in a considerable gain in speed at the expense of detail solvent effects.

Two major methods have been developed in recent years: The Monte Carlo method (MC) [12], 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 which is moved along an also randomly chosen vector. On the basis of the total energy of the system the Metropolis algorithm [12] is used to decide whether the new state is added to the chain or the old state counts again. This Markov chain is to converge to the thermodynamic equilibrium for some million states. Structural and thermodynamic properties of the investigated substance can easily be obtained. With a dynamic interpretation of the random movement of the particles (random walk) it is also possible to simulate non-equilibrium processes.

The MD method is based on numerical integration of Newton's equation of motion. During each time step, the interaction forces between the involved particles in a simulation box must be determined. After integration, the velocity of each particle is known and new atomic 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, dynamic and thermodynamic properties can be evaluated.

1.3.2 Potential Function

The key issue in modelling a molecular system is the choice of the potential function, also called the force field. Generally, the potential V of a system containing N particles consists of N additive partial potentials, each of them representing an interaction of a certain order. The first order interaction V_1 models the external field which is exerted on all particles. The most important second order potential V_2 is the pair potential standing for the interaction of all distinct particle pairs. The pair potential depends only on the distance between two particles and typically represents about 90% of the total energy in a many-body system. The third order term, involving triplets of particles, may not be neglected in simulations of liquids because its average energy contribution is around 10%. All higher order terms are expected to be small and usually are not taken into account. The triple-body interaction as a non-additive term in the potential function normally is not simulated explicitly (because the calculation of sums over triplets of particles is very expen-

sive in the sense of computational cost). Instead, the pair potential is replaced with the so-called effective pair potential V_2^{eff}, representing all multi-particle effects. Then, the overall potential reads

$$V(\mathbf{r}) = \sum_{i}^{N} V_1(\mathbf{r}_i) + \sum_{i}^{N} \sum_{j>i}^{N-1} V_2^{\text{eff}}(r_{ij}) \tag{1.8}$$

with r_{ij} denoting the distance (not the connecting vector) of the connecting vector of a particle pair (i,j). The notation of the effective pair potential summation should indicate that only distinct particle pairs (i,j) are considered.

To illustrate the contribution of multi-particle effects, fig. 1.1 shows the effective pair potential as well as the measured pair potential if only two particles are considered. The pair potential of two argon atoms is drawn as a solid line, whereas the dashed line is the effective pair potential as it would be used in a computer simulation of liquid argon ($N \gg 2$).

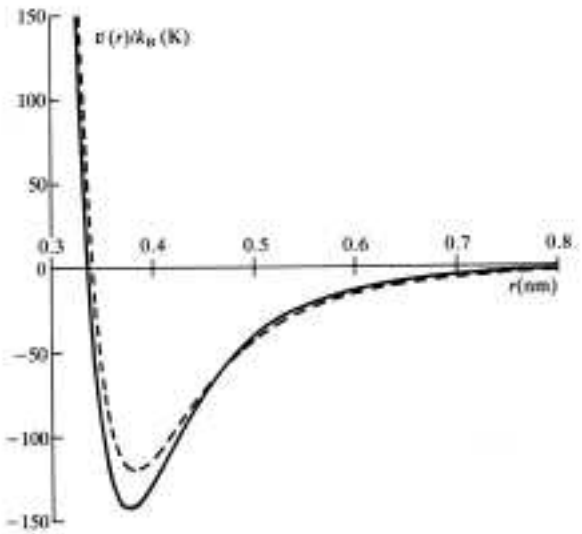


Figure 1.1 Normalised Argon pair potential: measured pair potential of two argon atoms (solid line), and effective pair potential approximated with the Lennard-Jones potential (dashed line) as it would be used to simulate liquid argon (picture source: [1]).

The potential as shown in fig. 1.1 is characteristic for intermolecular interactions: at very short distances (0.3 - 0.35nm), the steeply rising part indicates a strong repulsion or resistance to compression due to the nonbonded overlap between electron clouds. The negative sharp bend is the cohesian attraction in condensed phases (van der Waals force). The attractive tail consists of two parts, the fast declining ($\sim r^{-6}$) van der Waals force, and a long-range Coulomb contribution ($\sim r^{-1}$), which appears only for charged species and affects the potential even at longer distances (up to 2nm and more). Liquid argon is uncharged, charged species are for example ions or polarised molecules.

Staying at the simple level with only nonbonded interactions or one-atomic liquids, the most widely used effective pair potential approximation is the Lennard-Jones (LJ) potential (1.9). The LJ-potential energy depends only on the magnitude of the pair separation r_{ij} of an atom pair (i,j):

$$V_2^{\text{eff}}(r_{ij}) \approx V^{\text{LJ}}(r_{ij}) \;=\; 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{1.9}$$

The parameters ε, σ are determined by experimenting (measurement).

If a system contains multi-atomic molecules another problem arises: in addition to nonbonded (intermolecular) interactions covalent bonds (intermolecular) must be treated. Molecules typically 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. Although (1.8) is applicable to multi-atomic molecules, in most cases a separated potential as in (1.10) is preferred. The potential is composed of nonbonded interactions V^{nonb} (van der Waals and Coulomb interaction) and interactions between covalently bonded atoms V^{bon} (exchange of valence electrons). V^{special} may represent the already mentioned external force or friction effects at atomic level.

$$V(\mathbf{r}) \;=\; V^{\text{bon}}(\mathbf{r}) + V^{\text{nonb}}(\mathbf{r}) + V^{\text{special}}(\mathbf{r}) \tag{1.10}$$

The Lennard-Jones potential is mostly used as nonbonded interaction model. Models for covalent bonds may be rather complicated depending on the complexity of the molecule and the desired accuracy of the simulation. Good models take into account bond angles, bond stretching, dihedral angles, etc. Because the number of bonds in molecular systems is small (~N) compared with the number of nonbonded interactions (~N^2), much effort has been spent to keep nonbonded models as simple as possible, whereas the computation time of bonded forces is really negligible even with very accurate models.

Modelling the potential with a generalized pair potential—equation (1.10) combined with Lennard-Jones potential—is also known as the *central-force* model. This generalized pair potential includes both intra- and intermolecular interactions. The central-force method requires to solve as many differential equations as there are atoms in the system. The major drawback of this model is as follows: the intramolecular oscillation of most molecules is faster by about one order of magnitude compared with intermolecular motion. Assuming the same time steps of equal duration for all interactions, most time is spent by calculating unnecessary nonbonded interactions, only to keep track of the fast intramolecular motions.

1.3.3 Long–Range Interaction

To reduce the nonbonded computational demand, most Molecular Dynamics programs employ a *pairlist* paradigm [1]: the pairlist contains all nonbonded particle pairs whose connecting vector is shorter than a certain cutoff radius r_c. According to Newton's law of pairwise forces, only distinct pairs have to be taken into account. The interaction is then calculated only for those pairs in the pairlist. Due to the inertia of liquids it is not necessary to re-construct the pairlist each time step. Depending on the cutoff radius, a new pairlist is reconstructed every five to ten time steps.

A system with ionic charges introduces enough electrostatic potential that the cutoff radius r_c must be at least 2.0nm for moderate accuracy, which is often far away of maximum computation time constraints. Fast methods for ion simulation with relatively small cutoff radii are the *Ewald*

Sum ([11], [51]) and the *PPPM method* (particle-particle–particle-mesh, [13] and [51]). Both methods split the Coulomb function into a short-range term which is best handled with pairlist concepts, and a smooth periodic long-range function which is well suited for solution by Fourier methods. The Ewald Sum performs an analytical Fourier transformation, while the PPPM method relies on the very efficient numerical Fast Fourier Transform (FFT).

If only partial atomic charges contribute to Coulomb forces, the electrostatic interaction may be transformed to be r^{-3}–distance dependent. Here the trick is that a group of atoms whose atomic charges adds up to zero is treated as a *charge group*. Then, the interaction between charge groups is of bipolar type ($\sim r^{-3}$) and the radius for which electrostatic interaction really contributes can be reduced significantly. However, the contribution of far-away atoms is relatively small and, more important, does not change very much from time step to time step. To take full advantage of this effect, some MD simulators offer a *twin-range pairlist method* (section 4.4, [26]): every five to ten steps the pairlist is constructed using the (short-range) cutoff radius r_c. Nonbonded interactions are evaluated every simulation step using this pairlist. An additional long-range cutoff radius r_{cl} defines the twin-range, an additional space between the two spheres ($r_{cl} \geq r_c$). Nonbonded interactions affecting particle pairs with a distance longer than the short-range cutoff and smaller than the long-range cutoff are evaluated only if the pairlist is updated. These long-range interactions are kept constant between two pairlist updates. Nevertheless the long-range cutoff lies between 2.0nm and 3.0nm if ions are in the system and between 1.0nm and 1.6nm without ions. To minimise distortion effects at the cutoff boundary, the interactions beyond the cutoff may be approximated using the Poisson-Boltzmann reaction field method [1].

1.3.4 Periodic Boundary Conditions

Most simulations are done with *periodic boundary conditions* to reduce surface and edge effects. With periodicity, the simulation box is replicated throughout space to form an infinite space filling lattice. Fig. 1.2 sketches a two dimensional example of a solute molecule consisting of two charge groups dissolved in water.

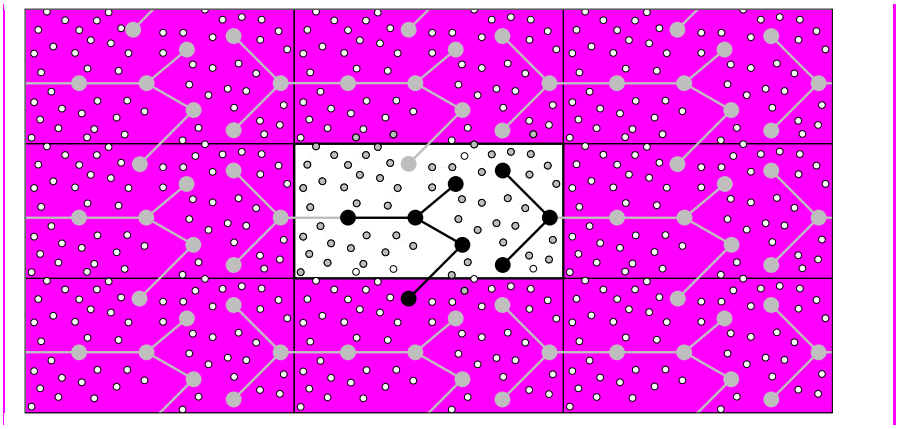


Figure 1.2 Periodic boundary conditions and charge groups: central computational box with eight periodic images of itself in the two dimensional space.

If a particle moves in the central computational box, its periodic image moves exactly in the same way in the neighbouring boxes. Consequently, if a particle leaves the central box, its image will enter through the opposite face.

It is proven in [5] that in three dimensional space there are five convex box types that can be stacked in a space filling way, i.e. that there are five possible types of boxes which may serve as a computational box. We refer to [5] for a complete discussion of all these geometries.

1.3.5 QMD Methods

The classical equations of motion are replaced (or supplemented) with quantum mechanical or quantum statistical methods to describe the quantum mechanical influences. Several QMD methods are currently used: the path-integral method [26], the density-functional method (Car-Parrinello [35]), the usage of the Wigner function [11], and semi-classical Molecular Dynamics with Gaussian wave packets [1].

With the path integral simulation technique quantum statistical mechanics is taken into account through a modification of the standard MD interaction potential. The method is based on an isomorphism between a classical system of rings, consisting of particles connected by harmonic oscillators, and the path-integral quantum mechanical description of the quantum system.

For almost classical systems the use of the Wigner function is appropriate. The function is defined as the Fourier transform of the density matrix containing the generalized coordinates and momenta. One approach is to model the quantum deviations as a supplementary term within the Hamilton function. These quantum corrections have been derived analytically up to the third order. Other methods allow the simulation of the quantum corrections.

Most recently approaches try to model quantum systems with one or a few quantum particles within a classical environment. Such systems may be simulated by solving the Schrödinger equation directly for the quantum particles, while performing regular MD simulation for the rest of the system ([31], [32]). Concrete implementations apply additional pairlists for the quantum particles.

1.4 The GROMOS96 Implementation

GROMOS is a general-purpose Molecular Dynamics computer simulation package for the study of biomolecular systems. Its purpose is threefold: 1) Simulation of arbitrary molecules in solution or crystalline state by the method of Molecular Dynamics (MD), stochastic dynamics (SD) or the path-integral method; 2) energy minimisation of arbitrary molecules; 3) analysis of conformations obtained by experiment or by computer simulation.The GROMOS suite includes the program *promd*, used to run a MD or SD simulation for one or more solute (protein) molecules plus solvent molecules. *promd* accepts as input simulation control parameters, molecular topology information, initial coordinates and (optionally) restraint and perturbation information. On completion, *promd* provides final atom coordinates and (optionally) MD trajectory coordinates, velocities, and energies.

1.4.1 Treating Boundaries

Simulations may be run in vacuum without periodicity, or with periodic boundaries using a rectangular, truncated octahedron, or monoclinic simulation box. The truncated octahedron is an octahedron fitted in a cube with edge length a such that the resulting volume is $a^3/2$. The motivation to implement this complicated shape in a simulation environment is to reduce the system size for spherical solutes. A monoclinic simulation box is a rectangular box with oblique angles. Monoclinic boxes are best suited for crystalline simulations. Within the thesis, rectangular and truncated octahedron boxes are further investigated in section 4.1 (neighbour searching).

GROMOS applies a charge group pairlist and supports the twin-range method. Nonbonded interactions are calculated either for all atoms of a charge group or for none. While a charge group moves outside the simulation box and periodic boundary corrections apply, the charge group as a whole is moved to the opposite side. Though it is possible that some atoms still reside outside the simulation box after the boundary correction took place (fig. 1.2). For example, this is the case for one atom of the five-atomic solute charge group in fig. 1.2. To ensure that one charge group does not have the same neighbour twice (once in the central computational box, and once as a mirrored particle), the *nearest image convention* restricts the cutoff radius to a certain size. In case of a rectangular box, the maximum cutoff is the half of the smallest side length.

Within GROMOS, *nonbonded* and *bonded* interaction types are separated. The Lennard Jones interaction potential model and a correction term contribute to the *nonbonded* part (1.26), whereas another model for covalent bond forces specifies the *bonded* part. For solvent, all intramolecular atomic interactions are excluded from nonbonded summation. Thus, the solvent nonbonded atomic interactions are calculated only between charge groups according to the pairlist.

1.4.2 The MD Algorithm

Both MD and SD simulation algorithms use the Leapfrog integration scheme. Additional constraints e.g. for bond lengths are handled with the SHAKE method. The algorithm which is presented here refers to the GROMOS96 time stepping MD simulation algorithm, however, not all formulae are given and the perturbed case is not treated at all. Formulae are given on the one hand to explain the force field which is of central interest, on the other hand to provide the basis from which computational models are derived in section 3.2.4. For a complete description of the algorithm, the Leapfrog and SHAKE techniques refer to [22], [26] and [60].

To avoid a naming confusion, small variable names, e.g. **r** and **v** refer to atom position and velocity vectors, whereas capital names like **R** and **V** refer to position and velocity vectors of groups of atoms or molecules. The same naming convention applies for the masses (M, m). If V is not bold, it is the potential V(**r**). Now, time dependences are included in the formulae.

Preconditions

1. Read molecular topology, initial coordinates and the simulation specification file. Read in optional data as specified: constraints, velocities, simulation box type and size, etc.
2. Calculate the solute and solvent kinetic energies (1.11) according to point two of the algorithm (below) and the corresponding temperatures. The initial time step is zero.

$$K = E_{kin}^{Solute}\left(t - \frac{\Delta t}{2}\right) + E_{kin}^{Solvent}\left(t - \frac{\Delta t}{2}\right) \tag{1.11}$$

3. Check the positions $\mathbf{r}(t)$ satisfying the constraints, and that the velocities $\mathbf{v}\left(t-\frac{\Delta t}{2}\right)$ are zero along the constraints.

Algorithm

1. Put solvent molecules and solute charge groups back into the simulation box if necessary.
2. Calculate the kinetic energy of the solute:

   Preparation: calculate the centre of mass velocity $\mathbf{V}_\alpha$ and the mass M_α of all solute molecules $\alpha = 1, \ldots, npm$, with npm the total number of solute molecules:

$$\mathbf{V}_\alpha\left(t-\frac{\Delta t}{2}\right) = \frac{1}{M_\alpha}\sum_{i=1}^{N_\alpha} m_{i\alpha}\cdot\mathbf{v}_{i\alpha}\left(t-\frac{\Delta t}{2}\right) \tag{1.12}$$

$$M_\alpha = \sum_{i=1}^{N_\alpha} m_{i\alpha} \tag{1.13}$$

   The double index $i\alpha$ denotes atom i of solute molecule α which consists of N_α atoms. Equation (1.14) delivers the atomic velocities relative to the molecular centre of mass $\mathbf{v}_{i\alpha}^{int}$, where 'int' indicates that this velocity is relative to the centre of mass.

$$\mathbf{v}_{i\alpha}^{int}\left(t-\frac{\Delta t}{2}\right) = \mathbf{v}_{i\alpha}\left(t-\frac{\Delta t}{2}\right) - \mathbf{V}_\alpha \tag{1.14}$$

   Now the energies: using (1.12)–(1.14), calculate the solute molecular centre of mass translational ('tr') kinetic energy and the solute internal ('int') and rotational ('rot') kinetic energy:

$$E_{kin}^{tr,Solute}\left(t-\frac{\Delta t}{2}\right) = \frac{1}{2}\cdot\sum_{\alpha=1}^{npm} M_\alpha\mathbf{V}_\alpha^2\left(t-\frac{\Delta t}{2}\right) \tag{1.15}$$

$$E_{kin}^{int,\,rot,Solute}\left(t-\frac{\Delta t}{2}\right) = \frac{1}{2}\sum_{i=1}^{N_{Solute}} m_i\mathbf{v}_i^{int^2}\left(t-\frac{\Delta t}{2}\right) = E_{kin}^{Solute}\left(t-\frac{\Delta t}{2}\right) - E_{kin}^{tr,Solute}\left(t-\frac{\Delta t}{2}\right) \tag{1.16}$$

   with N_{Solute} the total number of solute atoms.
3. If the Virial (1.20) is to be calculated, calculate the centre of mass positions of solute molecules:

$$\mathbf{R}_\alpha = \frac{1}{M_\alpha}\sum_{i=1}^{N_\alpha}\mathbf{r}_{i\alpha}\cdot m_{i\alpha} \tag{1.17}$$

   For solvent molecules, the centre of mass is the position of the first atom of the molecule.

   Calculate the atomic positions relative to the centre of mass $\mathbf{r}_{i\alpha}^{int}$ for solute molecules according to (1.18) and for solvent molecules respectively.

$$\mathbf{r}_{i\alpha}^{int}(t) = \mathbf{r}_{i\alpha}(t) - \mathbf{R}_\alpha(t) \tag{1.18}$$

4. Calculate a pairlist if required. The pairlist contains all charge group pairs in consideration of a cutoff criterion.

5. Calculate the unconstrained forces from the potential energy function $V(\mathbf{r}(t))$ according to (1.7) for all atoms $i = 1, \ldots, N$ in the system:

$$\mathbf{f}_i(t) = -\frac{\partial V(\mathbf{r}(t))}{\partial \mathbf{r}_i} \tag{1.19}$$

The potential is defined in equations (1.25) and (1.26). At the same time, calculate the *molecular Virial* $\Xi(t)$ if required. The summation in (1.20) runs over all pairs of atoms that are selected by the cutoff criterion used, and for which atoms i and j are in different molecules α, β; this is symbolised in the relation $i\alpha < j\beta$. The connecting vector $\mathbf{r}_{i\alpha j\beta}$ of such an atom pair is the nearest neighbour image vector, if periodic boundaries are applied. The subscript '$i\alpha j\beta$' indicates "different atoms in different molecules".

$$\Xi(t) = -\frac{1}{2} \sum_{i\alpha < j\beta}^{N} [\mathbf{r}_{i\alpha j\beta} - \mathbf{r}_{i\alpha}^{int}(t) + \mathbf{r}_{j\beta}^{int}(t)] \cdot \mathbf{f}_{i\alpha j\beta}(t) \tag{1.20}$$

6. If required: Calculate the volume $V_b(t)$ and the pressure $P(t)$ of the box; write the positions $\mathbf{r}(t)$, the velocities $\mathbf{v}\left(t - \frac{\Delta t}{2}\right)$ and box dimensions to trajectory files.

The rule that the average of the kinetic energy in a system equals the *molecular Virial* is used to calculate the pressure of the system:

$$P(t) = \frac{2[K - \Xi(t)]}{3V_b(t)} \tag{1.21}$$

7. Determine the new unconstrained velocities (integration using SD or MD Leapfrog algorithm) for all atoms i in the system:

$$\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}_i\left(t - \frac{\Delta t}{2}\right) + \frac{1}{m_i}\mathbf{f}_i(t)\Delta t \tag{1.22}$$

8. Determine the new translational centre of mass velocities of solute molecules $\mathbf{V}_\alpha\left(t + \frac{\Delta t}{2}\right)$ and the atomic velocities relative to the centres of mass $\mathbf{v}_{i\alpha}^{int}\left(t + \frac{\Delta t}{2}\right)$ by using equations (1.12) and (1.14) again.

9. Scale the new velocities $\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right)$ according to a optional temperature bath.

10. Calculate the new unconstrained positions

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i\left(t + \frac{\Delta t}{2}\right)\Delta t \tag{1.23}$$

11. Make the positions $\mathbf{r}_i(t + \Delta t)$ satisfy the constraints using the SHAKE algorithm.

12. Calculate the constrained velocities by replacing the unconstrained ones:

$$\mathbf{v}_i\left(t + \frac{\Delta t}{2}\right) = \frac{\mathbf{r}_i(t + \Delta t) - \mathbf{r}_i(t)}{\Delta t} \tag{1.24}$$

13. Calculate the new solute kinetic energy $E_{kin}^{Solute}\left(t + \frac{\Delta t}{2}\right)$ and the new solvent kinetic energy $E_{kin}^{Solvent}\left(t + \frac{\Delta t}{2}\right)$ by using equations (1.15) and (1.16) again. Determine the new kinetic energy K of the system (1.11).

14. Write additional data to trajectory files and/or standard output/error if required.

15. Scale atomic positions and the box lengths with the pressure coupling scaling factor if required. Scale reference positions of positionally restrained or constrained atoms.

16. Increase the time $t_{n+1} = t_n + \Delta t$ and the step number by one and continue with step one of the algorithm.

1.4.3 The Force Field

The calculation of the unconstrained forces as in step five of the algorithm is the major task in a MD simulation. In principle, the interaction potential describes the whole dynamical model of the atomic system where the choice of a certain force field is coincident to the fixing of the potential. Like the majority of all MD simulators GROMOS96 implements the concept of potential separation distinguishing the standard physical atomic interaction separated into bonded and nonbonded terms and a non-physical special term, see (1.10). The special potential is for particular purpose, e.g. restraining functions. According to the molecular model [26], the bonded potential is further separated:

$$V^{bon}(\mathbf{r}(t)) = V^{bond}(\mathbf{r}(t)) + V^{angle}(\mathbf{r}(t)) + V^{har}(\mathbf{r}(t)) + V^{trig}(\mathbf{r}(t)) \tag{1.25}$$

Thus, GROMOS applies the central-force model to determine the potential. Bonded contributions come from the covalent bonds itself, the bond-angles, harmonic or improper dihedral angles, and trigonometric dihedrals. For the description of the bonded contributions refer to [26]. To overcome the problem of very small time steps Δt due to fast oscillations in bonded terms, distance constraints may be applied along covalent bonds. With this technique, which is called *constraints dynamics*, Δt is determined by the relatively slow intermolecular interaction. The fast oscillations are handled with the SHAKE algorithm.

Nonbonded forces are calculated for all distinct charge group pairs (nb pairs) under consideration of a cutoff criterion. To model nonbonded forces, GROMOS makes use of a potential composed of a standard Lennard-Jones effective pair potential and a reaction field correction term (1.26).

$$V^{nonb}(\mathbf{r}(t)) = \sum_{(nb\ pairs)} [V^{LJ}(\mathbf{r}(t)) + V^{CRF}(\mathbf{r}(t))] \tag{1.26}$$

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

Following the charge groups concept, the Coulombic interaction is included in (1.26). The reaction field correction prevents distortions due to the cutoff criterion. The parameters C_{12}, C_6, C_{rf}, and R_{rf} (1.26) are called *force field parameters*. Additional parameters appear if atom i or j is *perturbed* which is the case in *free energy simulations* [26].

When simulating liquids the nonbonded interaction calculation is the most time consuming task. Therefore, GROMOS differentiates two kinds of molecules, namely *solute* and *solvent* molecules. This naming convention is based on historical grounds but has been retained, although today it is possible to specify any collection of molecules including solvent in the *solute* part. The implementation of the nonbonded interaction calculation makes use of different specifications: the function for *solvent* restricts the available features such that its implementation is fastest while the *solute* part supports all available GROMOS features. These limitations do not go to the extent that for each kind of solvent a dedicated optimised subroutine is provided. The most important restrictions are (affected functions enumerated in brackets):

- A solvent molecule must be rigid. The internal structure is maintained by applying distance constraint forces between its atoms (SHAKE algorithm, bonded forces).
- A solvent molecule consists of exactly one charge group and the position of the first atom in a solvent charge group is taken to represent the position of this charge group (preparing the Virial, pairlist).
- Distance restraining cannot be applied to atoms of a solvent molecule (nonbonded forces).
- Solvent molecules may not be perturbed (nonbonded forces).

The restrictions do have significant implications on the complexity of the implementation. The solvent-solvent part of the nonbonded interaction calculation is simple, see (1.26), while particle pairs including solute charge groups are much more complex to calculate, mainly caused by the code for the perturbed case.

1.5 Parallel GROMOS: Motivation and Main Goals

1.5.1 Related Tools

GROMOS has been developed by W.F. van Gunsteren and co-workers since 1978 at Harvard University (USA), the University of Groningen (The Netherlands) and ETH Zürich (Switzerland). The latest official release (GROMOS96: [26] and [60]) is written in standard FORTRAN 77 with the only exception that *include files* are used, mostly consisting of global variables and constants. Several hundred licences[1] in over 40 countries and the availability of the source code for research purposes promotes the extendability significantly. GROMOS codes can be run on various machines ranging from PCs to super-computers. Special nonbonded force calculation subroutines (High Performance FORTRAN–HPF) for Cray computers and for the SGI Power Challenge are available. Competing with GROMOS are tools such as AMBER ([8], [57]) and CHARMM [34] (Chemistry at HARvard Macromolecular Mechanics), which belong to the same class of applications and offer similar functionality.

AMBER has been developed at UCSF (University of California, San Francisco) in collaboration with other universities and companies. Originally derived from the same source (AMBER is a further development of an early version of GROMOS) the software architecture and fundamental

1. For more information, see the Gromos homepage at http://igc.ethz.ch/gromos/ or contact Biomos b.v., Laboratory of Physical Chemistry, CH-8092 ETH Zürich, Switzerland.

functions are very similar. Additionally, AMBER provides a Particle-Mesh Ewald (PME) method to treat long-range forces and a quantum mechanical simulation method, where one part of the system is treated quantum mechanically, and the rest classically. However, SD simulation and path integral method, both of which are features of GROMOS96, are not provided by AMBER.

Initially published in 1982 as "A Program for Macromolecular Energy, Minimization, and Dynamics Calculation" [34], the CHARMM Molecular Dynamics simulation program has been consequently developed further. Similar to the AMBER package, the current version of CHARMM incorporates a fast multipole method based on the Ewald sum and a quantum mechanical simulation method. The latest versions were written in FORTRAN (machine dependent code in C) and provide some visualisation possibilities for X-Windows systems. A parallel version of CHARMM, based on a replicated data model and PVM interprocessor communication, is available for various machines (PVM: parallel virtual machine). Like GROMOS, CHARMM has the possibility to run MD simulations in four spatial dimensions.

1.5.2 GROMOS96P Design Goal

In general, molecular systems with an overall complexity of about 50,000 to 100,000 atoms are considered. Such systems typically consist of one or more solute molecules surrounded by a liquid (solvent). In each time step Newton's equations of motion are solved by numerical integration of the intra- and intermolecular atomic forces. Despite the usage of a pairlist combined with reaction field correction techniques, one time step of a 36,000 atoms simulation takes 30 seconds on a currently available high performance workstation ([42], and table 3.2). A ten nanosecond simulation would thus take about two days, whereas for a breakthrough MD research simulations over microseconds are required. The great demand for accelerating techniques (new algorithms, special purpose hardware, parallelisation) is therefore a big challenge for developing dedicated parallel systems.

In order to speed-up MD simulations of liquids, general acceleration techniques have been developed in the recent past: efficient algorithms, parallel simulations on general purpose computers and dedicated hardware accelerators. The gain obtained from more efficient algorithms is obvious and indisputable, and the choice of an algorithm is a compromise between accuracy and calculation speed. One example of an algorithmic improvement is the usage of efficient neighbour searching methods [64] to build the pairlist. Most recently published parallel software implementations are often aimed at multi-million particle systems [49] or massively parallel machines. Of course, the basic concepts in parallelising MD code (section 4.2) are more or less the same and undisputed. Dedicated parallel hardware architectures are often developed for astrophysical systems with millions of particles [52]. Corresponding software for MPPs (massively parallel processors) is optimised for thousands of processors, parallel input and output (I/O) and irregular particle distribution ([28], [62]). Highly specialised hardware accelerators for MD simulations (MD-Grape [40] and MD Engine [63]) with ASICs for force calculation do not support special MD functions like energy summation, Virial calculation, free energy calculation, never mind Newton's Third Law of pairwise forces [40]. In addition, parallel MD systems are often targeted towards high performance and do not provide full production code functionality [54][36][47].

Already existing parallel versions of GROMOS from other sources suffer from reduced functionality: EulerGromos [36] and UHGROMOS [47], which do neither support parallel simulation of a truncated octahedron (section 4.1.3) nor the twin-range pairlist method (section 4.4). Parallel code which is delivered with the standard GROMOS96 distribution is not sufficient for the following reasons: a shared-memory version implemented in High Performance FORTRAN (HPF) scales rather badly and may not fulfil our performance demands. Supporting in addition a vectorized version, which is outdated anyway, leads to large maintenance costs surpassing the capabilities of a lab in a university environment. To overcome these problems and simplify the integration of new features like QDGROMOS ([6], [32]), an extension to simulate some particles quantum dynamically within a classical environment, the development of GROMOS96P—the new parallel GROMOS based on the '96 release—is inevitable. The expectations from this new release concerning both software *and* hardware issues are as follows:

- The main goal of our research is to speed-up classical MD simulations of liquids by at least *one order of magnitude*. The speed-up is defined as the ratio of the execution time of the sequential GROMOS96 official release to the execution time of the new parallel GROMOS96P, both of which running the typical performance benchmarks *ThrG2* and *Thr2* (appendix A).
- This speed-up is to be achieved either with an arbitrary parallel architecture or with a dedicated scalable coprocessor for high-end workstations.
- The overall hardware cost may not exceed the amount of about 20,000 USD.
- In order to ensure the acceptance within the GROMOS community, the new software must also run on all currently supported platforms (GROMOS96 release).
- Avoid many versions of the same function in order to improve the *maintainability*, e.g. two versions of an interaction calculation function, one specialised for a shared memory architecture and the other for a vector machine.
- The GROMOS96P environment should derive different program versions for different parallel and single processor architectures *from the same source code*.
- This automatically improves *extendability*: if a new function is to be added, the programmer provides one version of the new function which is inserted into the source code at one dedicated point. So the user is not hold up by programming special code for a certain architecture.
- The new system must be *scalable*: by adding more hardware, an enhanced speed-up of about 20-40 is to be reached for reasonable molecular problem sizes.
- GROMOS96P software *shall not be slower* than GROMOS96 on a single processor workstation.

Since one of the main goals was *platform independence*, the new parallel version of the software had to be developed independently of the targeted parallel architecture. Thus great attention was set on drawing a sharp distinction between the new GROMOS96P parallel software and supplementary libraries. As it is naturally the case with any hardware extensions for PCs and workstations, a dedicated MD coprocessor would also come with its appropriate driver software.

All the mentioned goals should be achieved by leaving the GROMOS96 look and feel untouched, changing neither the simulation setup nor the format of input and output data. This point also implies the downward *compatibility*. As stated in the list above, the intention was to improve many things without of losing even one of the achievements GROMOS96 provides. Therefore, the new system must provide full production code capability and the same *numerical accuracy* as GROMOS96. In addition, the *portability* of the new code must be retained, as well as the esteemed *ease of use* of GROMOS96.

Fortunately, to achieve the required speed-up and to stay within reasonable bounds regarding the parallel software complexity it is not absolutely necessary to parallelise the complete GROMOS96 package (section 3.2.3). The focus was on parallelising the most computation intensive parts of the classical MD algorithm by providing full production code functionality.

Naturally the parallel architecture, on which the new parallel GROMOS96P will be implemented, was not obvious in the beginning. Therefore, a novel design space exploration method has been developed to find the optimal trade-off between time-to-market, cost and performance [44]. To achieve shortest development time, the latest Hardware–Software Codesign technologies [15] and new approaches in system synthesis [17][16] were applied.

CHAPTER 2

Methodology

The most important conditions for all kinds of product development are: being on time, to be as cheap as possible and as good as required. The industry has started looking at the product development cycle comprehensively, to reduce the design time to be competitive in the time-to-market race. Recent trends show that time-to-market is at least equally, if not more important, than the performance or the cost of the product. Experience in the development of electronic systems has shown that a good design trade-off is found very often in a mixture of hardware and software. Providing mixed hardware–software implementations using *synthesis techniques* lead to the co–synthesis approach, where the prefix 'co–' means 'concurrent'. To achieve the shortest design time along with minimal cost and maximum performance, it is indispensable to apply synthesis-oriented methods. High level modelling of systems and synthesis techniques allow systematic and rapid evaluation of implementation alternatives, which is a prerequisite for efficient design space exploration.

This chapter starts with an outline of what is currently understood by the term "Hardware–Software Co–Synthesis" and gives an overview of recent developments in high-level synthesis. For a more comprehensive discussion of the subject we refer to [15], [16] and [17]. In what follows, three main topics are selected and worked out in more detail: 1) how systems are modelled 2) design space exploration using system synthesis methods and 3) software synthesis. It is not intended to present here a thesis about design methodology in general, but to introduce the methods and techniques which have been used in the Molecular Dynamics project. This chapter may be treated independently but constitutes the theoretical foundation for subsequent chapters.

2.1 System–Synthesis

2.1.1 Levels of Abstraction

Using formal methods in system design has several advantages: a *formal specification* can be verified, semantic checks are easily implemented, and lastly, correctness and reliability can be guaranteed to a large extent. All these qualities have promoted the use of formal methods to the extent they are used today. A *model* is a formal description of a system or part of a system. Deliberately considering only special properties of the system for a specific model is called *abstraction*. An

abstraction is a special view of a system and, depending on the modelled characteristics, the resulting formal description is either a high level model, a piece of executable assembler code or something between.

In the behavioural domain what a design *does* is of central interest, not how it is built. This domain is in contrast to the structural view, which is described in the next paragraph. High level models describe the interplay with the outside world and the behaviour of the whole system under consideration. In addition to the *behavioural model* and the *interface specification*, a complete *behavioural specification* for a certain abstraction level may also include a description of the *constraints* imposed on the system. Starting with such a behavioural top level system specification, a number of design decisions are required to accomplish system partitioning into subsystems. The design decisions concern the target architecture, and the partitioning determines the mapping of parts of the behavioural model (*functions*) to hardware components. The next step includes the implementation of associated subsystems, whose behavioural models mostly are derived directly from the system level description by adding an additional level of detail. Implementing a subsystem means partitioning the specification into hardware and software. The hardware part defines the architecture of a subsystem and corresponding behavioural models describe the properties of each component. Typical components on this level of abstraction are communication channels, computational nodes (e.g. processors), etc. Hardware components are to be refined to the RTL (Register–Transfer Logic) level. Typical behavioural models on this level consist of a data path structure and a state machine specification to control that data path. Finally, the logic level is described with boolean expressions, and the next refinement step may consider circuit descriptions of physical gates (e.g. transistor behaviour: timing and other properties). The abstraction levels of the software part are more intuitive: the software is decomposed into processes, modules, blocks and subroutines, all of which are implemented using a high level programming language. Further refinement steps include the compilation process which results in assembler code, which is then translated into executable binary code.

As mentioned above, on most abstraction levels the system is represented from different views, each view providing either relevant information for the next refinement step (top-down) or the interface to the next higher level of abstraction (bottom-up). The *behavioural view* represents a certain abstraction level with a set of functions independent of their implementation. The appropriate *structural view* stands for one specific implementation of the behavioural model at the same abstraction level. So the structural representation bridges the behavioural specification and possible physical implementations. For example, the structural view of a behavioural subsystem specification is a block diagram consisting of processors, memory, communication channels, etc. As another example, a graphical VHDL[1] specification of a finite state machine (behaviour) is refined by generating a concrete implementation (structure) for a specific target, e.g. a programmable logic device (PLD).

Of course designing complex systems is not just a sequence of many refinement steps starting from a system level behavioural description, several iterations between different levels of abstractions are generally inevitable. Because the focus is on short development time, there is an increasing need for computer aided design (CAD) tools which partially automatise these iterations. In addition, modern systems are not developed following a strict top-down approach, as a higher

1. VHDL=VHSIC–HDL, Very High Speed Integrated Circuit Hardware Description Language, [46]

level behavioural specification may be based on existing lower level implementations. CAD tools should be able to consider that too. To make matters worse for tool implementors, alternative classifications of levels of abstraction are in common use [16].

2.1.2 System Synthesis: Definition and Motivation

Synthesis is the (semi-)automated translation process from a behavioural description into a structural description, e.g. between two adjacent levels of abstraction or between sights within the same level. The input for *system synthesis* is the behavioural specification of the whole system. Associated with the input specification are constraints like execution time (performance) or hardware cost. Several other factors may also affect synthesis results, namely the estimated development time for a certain solution, existing reusable approaches or products, and more. Under this point of view, system synthesis is a multi-objective optimisation problem. System synthesis resolves the following tasks:

1. A target architecture (structural view) is provided, consisting of hardware components.
2. The behavioural model is partitioned into single functions or function blocks.
3. The mapping of functions to hardware components is determined.

On the highest abstraction level possible target architectures cover the whole range, from client/server systems or workstation clusters over shared memory multiprocessors, specialised workstation architectures, embedded systems, to heterogeneous solutions. So the design space, which is defined as the space spanned by all possible implementations and all possible architectures, is very large.

For lower levels of abstraction the traditional capture-and-simulate design methodology has already been replaced with automated design tools. Especially for VLSI (Very Large Scale Integrated) circuit designs, behavioural compilers are available delivering the RTL data path structure as well as the corresponding state machine. Usually both input specification and delivered output of the behavioural compiler are represented in a hardware description language, for example VHDL. Within the software domain, synthesis is even more widespread: everybody is using a general purpose compiler to translate his source code written in a high level programming language into a machine executable program. Since synthesis on lower levels of abstraction is practised for quite some time, the wish is to move design automation to higher abstraction levels. Moreover, as the complexity of systems increase, the need for high level design automation will also rise, enabling very complex tasks like *design space exploration* to be (partially) automated. There are many reasons to move synthesis techniques to higher abstraction levels:

- Systems are *easier to understand* on higher levels of abstraction as the models contain only the most critical design parameters.
- *Fast design cycle*: to meet time-to-market constraints, CAD tools are essential. Even if the CAD tool is capable to synthesize only parts of a system, it is still a major gain in time, and verification for manually refined parts is still possible.
- For an increasing number of applications a suboptimal but synthesized implementation is more *cost effective* than designing or optimising manually for optimal results.

- *Faulty designs*: synthesis tools avoid iterations due to design errors. Provable correct results are guaranteed, because the synthesizer follows comprehensible transformation rules. The knowledge of these rules enables a formal verification of the results.
- *Design space exploration*: Using synthesis techniques, many design alternatives can be generated, quickly evaluated and assessed.

2.1.3 The Basic Synthesis Algorithm

Inferring from the previous section, the task of system synthesis may be understood as an optimisation problem. This is comparable with compilers, where different optimisation goals lead to either fastest implementations or smallest programs. A system synthesis tool, taking into account the constraint properties, and the design space as defined earlier, has to solve the optimisation problem to be able to provide at least one implementation proposal. After the optimisation goal is declared (e.g. minimal cost), the problem is solved by iteratively executing 1) *allocation*, 2) *binding* and 3) *scheduling*.

The *allocation* step selects an architecture, especially the number and kind of hardware resources like generic computational nodes, communication channels (network, buses), processors, ASICs (Application Specific Integrated Circuit), etc. This step obviously influences the cost–performance balance. Parts of the partitioned behavioural model are associated with allocated resources within the *binding* procedure. The *schedule* determines the sequence of execution of all functions of the behavioural model. This is done by defining time intervals for functions and communications: one specific task may start execution or data transportation only within its associated time interval. After the schedule is determined, the implementation proposal is checked against constraint violations. If it is found to be valid, synthesis was successful. It should be noticed that within this context, a valid implementation is still an abstract model for which it is expected that a physical implementation meeting the optimisation goal may be found.

Additional problems may arise if also political or other environmental authorities, e.g. a business policy, are affecting further project drafts. Design trade-offs are mostly not strictly based on synthesis results, but also on these informal influences and the designer's knowledge, e.g. about reusable systems or parts of a system. This kind of constraints are difficult or impossible to incorporate in a CAD tool, and due to the "human nature" of design decisions at this abstraction level synthesis tools are still rare, if at all available, and still need a lot of manual interaction. Nevertheless, synthesis results are often doubtful, and intuitive or heuristic design decisions lead to better implementations. Hence it is not surprising that CAD tools still cannot completely automatise the synthesis of complex systems, although high quality designs are generated for single tasks.

2.1.4 System Synthesis and Design Space Exploration

Most important design decisions at system level mainly determine cost and performance of an implementation. To explore the design space thoroughly, as many design alternatives as possible must be tested if they meet the design goals. The previous section has shown that generating appropriate implementation candidates may be automated by using system synthesis: different solutions are obtained by repetitively executing a synthesis step with modified parameters. From this point of view, design space exploration means iterating between adjacent abstraction levels or different sights of the same level. The iteration is terminated if at least one apparently suitable tar-

get architecture could be found. Using synthesis techniques, many design alternatives can be generated, quickly evaluated and assessed. Applied to design space exploration, this is a suitable aid to search target architectures. Combined with appropriate performance and cost estimation, faulty design decisions causing to conflicts with constraints may be detected without implementing them. But in most cases, synthesis is not the only source of implementation suggestions. The knowledge of the designer about existing third party implementations combined with creativity, fantasy and intuition is the second important source.

As a key prerequisite for system synthesis the top level behavioural specification must be provided in a formal representation with a specified syntax and semantics, requiring the specification language to be compatible with the synthesis tool. The specification model should be chosen such that the behaviour of the system (the algorithm, interfaces and constraints) may be described, along with the synthesis results and other implementation suggestions (structural view). The choice of the specification model and its language is one important issue, providing enough design alternatives another one, but successfully exploring the design space needs even some more tasks and methods:

- Most important is a *fast assessment* of the implementation proposals. Those are not only the results of system synthesis, but also other possible architectures.
- The assessment should take into account the cost of a binding and the performance of the schedule. The assessment must be fast or automated to allow many hypothetic system implementations to be compared with each other within reasonable time.
- Performance comparison requires an accurate *performance and cost estimation*. For that, suitable metrics must be found or developed.
- The specification model should serve to describe implementations for performance and other estimations as well. Thus the model must be able to describe architectures, algorithms or programs, and to consider constraints.

Once an adequate solution is found, the subsystems are refined concurrently. If a subsystem consists of hardware and software, codesign technologies are applied [17][15]. During the design process, when the functional mapping changes (modification of system partitioning), new functions are added (modification of the algorithm), or behavioural model parameters are modified (modification of constraints, optimisation goal, etc.), the current partitioning and its implementation must be verified. This iterative process ensures that the design goal with constraints is granted at any time.

2.2 Specification and Modelling

Models of algorithms and architectures are required to use synthesis methods, but also for manual refinement a good formal model is the base for the design and analysis of fast and portable parallel algorithms. The reason is because it is not enough to provide a lot of promising implementation variants, at least equally important are methods to estimate their cost and performance. Estimations are used to assess and compare the variants with each other. In particular, we are looking for a specification method which is best suited for design space exploration, especially in connection with the Molecular Dynamics project. The demands on a specification model are manifold, and for our purpose the following tasks must be covered:

- The system level behavioural specification describes an abstract model of the MD algorithm as introduced in section 1.4.2.
- The same specification model should be applicable to system level and subsystem level.
- A slightly more abstract behavioural view of the top level specification is used for system synthesis (section 2.3.2). This view incorporates only properties of the system which are valuable for the synthesis tool.
- The model must be suitable to describe parallel architectures (structural view) and parts of the algorithm (behavioural models of functions). This is required to allow accurate performance and cost estimation.

Following an overview of some common modelling and specification methods, the modelling techniques which were applied to the parallel GROMOS project are introduced.

2.2.1 Overview

The search for a suitable specification language is very much a subject of ongoing research, and the final choice of a specification method depends on many factors. The choice of a system or algorithm specification method depends on the one hand on the level of abstraction, on the other hand on the system itself ([15], [9]). Since it is very popular to use procedural hardware description languages like VHDL to specify hardware and software on intermediate abstraction levels, its usage for behavioural specification on system level is not convenient: VHDL is a complex, very general parallel language. A system synthesis tool coping with the whole complexity of this language is unimaginable, since the specification has to be analysed thoroughly. Additionally, it makes no sense to use such a complex language for specifying on system level, where the specification should be most abstract. As an advantage, VHDL specifications can be used throughout the levels of abstraction, for behavioural and structural descriptions.

Alternatively, other parallel languages are commonly used: Verilog [23], CSP (Communicating Sequential Processes) [19], Statecharts [18], SDL (Specification and Description Language) [4], and many more. Verilog is competing with VHDL and offers similar features, unfortunately the complexity is also comparable. The CSP language may be used as a programming language and for specifying hardware. Thus it is rather complex and supports a task level parallelism with message passing communication. Statecharts are mainly used for modelling reactive systems and might be a good choice for this domain. The advantage of CAD tools based on Statecharts is the possibility to generate either VHDL, Verilog or C code from any specification. SDL has been used mainly in the telecommunication field for protocol specification, but does not support programming constructs and provides no suitable output for further processing. All the mentioned languages do not support dataflow descriptions.

In contrast to procedural programming languages, data flow graphs (DFG, [17][48]) provide a data-driven representation with the advantage that multiple threads of execution are part of the model. Available tools usually produce VHDL, Verilog or C code as output. Supported specification languages are application specific and cover all common models as mentioned above. Keeping in mind that a specification language for system synthesis shall be found, domain-specific languages seem more appropriate. Domain-specific languages capture only the important properties of the current system and the level of abstraction for which they are designed. For example, to describe behaviour on logic level, a domain-specific language would allow only boolean expres-

sions and state diagrams. Similarly, a specification model for GROMOS would incorporate only operations which are needed by the synthesis tool, or operations to describe the behaviour required for performance estimation. Lastly, domain-specific languages may be easier to apply.

The top level behavioural specification, once established, mostly is the constant factor in the design process, and a structural model is what is needed next. Parallel algorithms running on parallel architectures demand corresponding models, and modelling parallel computation is more ambitious than sequential equivalents. A lot of simplistic computation models have been developed so far, most famous the PRAM (parallel random access machine) model and its derivatives. The original PRAM model [39] is rather unrealistic, because it assumes that all processors work synchronously and that interprocessor communication is free. Several variations of the PRAM model cover more practical cases, e.g. asynchronous execution and modelling of realistic communication channels. Nevertheless, because of the restriction to shared memory architectures, the PRAM model is not suitable for our purpose. Instead, the logP model [37], which follows the message passing principle for distributed memory systems, was further examined (section 2.2.4).

None of the mentioned models are absolutely convincing to be applicable to the GROMOS project. Therefore, a domain-specific specification model has been developed, in conjunction with a derivative of the logP parallel computing model. The remaining part of section 2.2 outlines this enhanced parallel computing model including architectures, communication and performance, and the algorithm specification. The presented graph model for behavioural system specification includes properties like function complexity, memory demand and I/O.

2.2.2 Parallel Performance Metrics

Metrics which capture processor characteristics in terms of the clock speed, the instruction execution speed (millions of instructions per second, MIPS), the floating point performance (millions of floating point operations per second, MFLOPS), and the execution time for standard benchmarks (e.g. the SPEC benchmark suite[1]) have been widely used in modelling uniprocessor performance. On the other hand, the hardware specification (processor, memory hierarchy, interconnection network) may never be a true indicator of the performance delivered by a parallel system. This is due to *communication* overheads and *algorithmic* overheads arising from the parallelisation process and (fig. 2.1).

The *available* speed-up is an ideal curve and determines linear scalability without loss. *Algorithmic* overhead is composed of remaining serial parts in the parallel program plus new code which had to be introduced to divide up the workload. The *achievable* performance of a parallel application is the theoretically reachable upper bound on an ideal machine. Communication overhead is separated into one part indicating the physical delay—bounded by the bandwidth of the communication subsystem (indicated as *communication* in fig. 2.1)—and another part representing the software overhead introduced e.g. by protocol stacks (indicated as *comm. overhead* in fig. 2.1). Measuring the performance of typical benchmark applications (*profiling*) on a real parallel architecture determines the *delivered* performance.

1. Founded 1988 by a small number of workstation vendors, SPEC has grown to become one of the most important performance standardisation bodies with more than 40 member companies. The SPEC benchmark suite covers a wide application area: Java-, CPU-, network-, parallel systems and www-benchmarks are freely available. Refer to: System Performance Evaluation Cooperative (SPEC) at http://open.specbench.org/.

Machine and application specific performance metrics provide detailed information not only about the overall performance of a parallel implementation but also about where performance is lost. Once the problem is identified, it is essential to find the individual application or architectural artifacts that lead to these bottlenecks and quantify their relative contribution towards limiting the overall performance. One concept to capture the growth of particular system overheads with respect to specific system parameters like network topology or the number of processors is to introduce overhead functions [61]. These overhead functions would appear in addition to functional nodes in a behavioural algorithm specification.

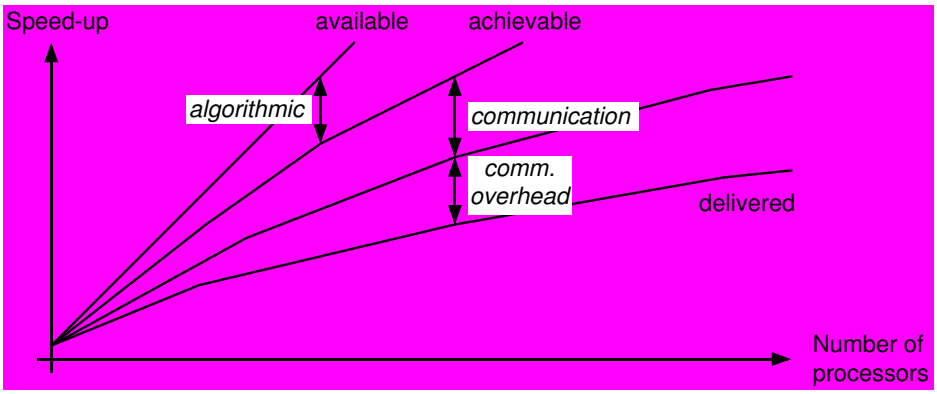


Figure 2.1 Parallel systems overheads

Performance measurement by *experimentation*, *analytical modelling*, and *simulation* are three well-known techniques that can be used for quantifying parallel computing overheads. Experimentation involves the implementation on a typical hardware and measuring the performance. Analytical models describe the architecture of the target hardware as well as the timing behaviour of complex application features with simple formulae. Simulation means that the application is not run on the real target machine but is simulated with a tool imitating the behaviour of the target hardware.

Evaluating the performance by experimentation of course is possible only if the design under test is accessible, which is not the case in the design space exploration phase where only models of possible implementations are available. The remaining alternatives are 1) implement and test all promising approaches of the exploration step, 2) simulate everything and 3) use estimations for assessing the quality of the design. The first two options correspond to *experimentation* and *simulation* as already defined, both of which might be expected as too expensive (cost *and* time) during the exploration phase. As a result of this, the simplicity of estimations combined with appropriate models surpasses the accuracy of simulations, since implementing a simulator is not a very simple task.

Consequently, analytical models as they were used to explore the design space for GROMOS96P (section 3.3) are discussed in more depth. Moreover, in section 3.2.2, the performance of the sequential GROMOS96 is measured through experimentation. A slightly different performance metrics as the one of fig. 2.1 is introduced in section 6.2, specialised to asses the GROMOS96P parallel implementation through experimentation. Performance estimation by simulation is not further examined.

2.2.3 Modelling Parallel Architectures

Historically, it has been difficult to develop a reasonable abstraction of parallel machines. The reason is that the machines exhibited such a diversity of structures represented by radically differentiated hardware organizations, including MIMD (multiple instruction–multiple data) and SIMD (single instruction–multiple data) machines, vector processors, systolic arrays, dataflow, shared memory, and message passing machines. Recent years of parallel machines development have brought some convergence towards systems with a familiar appearance: microprocessors with cache and DRAM (dynamic random access memory) main memory build one node of which many are interconnected with a highly reliable and fast communication network. Variations of this basic structure involve the communication interface and nodes consisting of more than one processor.

Standard microprocessor performance increases about 50–70% per year, and the memory capacity about 30–50%. Parallel machines must follow this curve of performance growth to remain competitive. Because the DRAM access time is not growing as fast as the operating frequency of microprocessors, sophisticated cache structures are required to bridge this timing gap. To follow this technological progress popular manufacturers of high performance parallel machines started to use off-the-shelf microprocessors or even full workstation nodes in their latest parallel computers. Future machines will be aimed at hundreds or thousands of 64-bit off-the-shelf microprocessors instead of machines consisting of millions of one-bit processors. As an example, IBM's PowerPC-based distributed memory scalable supercomputer RS6000/SP may be cited [53]. This architecture typically consists of shared memory nodes with up to four processors per node, which are interconnected by a switched high performance network; the system is extendable up to 512 nodes (appendix B). This trend towards fewer computational nodes has a significant impact on the design of parallel algorithms: they must be developed under the assumption of a large number of data elements per processor.

Network performance is also improving, but at a slower rate as processing power. Therefore, it must be assumed that latency, bandwidth and communication overhead will continue to be problems.

From the software point of view, no single programming methodology is becoming clearly dominant: shared-memory, message-passing and data parallel styles all have significant popularity. Thus, the requirements to a good computational model for parallel systems may be summarized as follows:

- The computational model should apply regardless of programming style.
- It should abstract the interconnection network such that only the performance characteristic is described but not the structure of the network.
- The user must be enforced to specify all system relevant aspects in a formal language, so automatised processing of a specification will be possible.

Block diagrams of the hardware (fig. 2.2, left) are widely used for documentation, presentation or explanation due to its intuitive characteristics. Because this pragmatic model is informal, it is not well suited as a model for system synthesis, analytical analysis or performance estimation of the GROMOS application. To overcome these drawbacks, the block diagram is transformed to a abstract graph model as in fig. 2.2 on the right, representing the same architecture. In the *archi-*

tecture graph $G_A(V_A, E_A)$, consisting of nodes $v_i \in V_A$ and edges $e_l \in E_A$, *computational nodes* (grey) replace processing nodes and *communication nodes* (white) model the interconnection of the components, with P_{tot} the total number of computational nodes or processing elements V_{PE} in the system, with $V_{PE} \subset V_A$ and $|V_{PE}| = P_{tot}$. Edges in G_A model directed links between computational nodes and communication nodes.

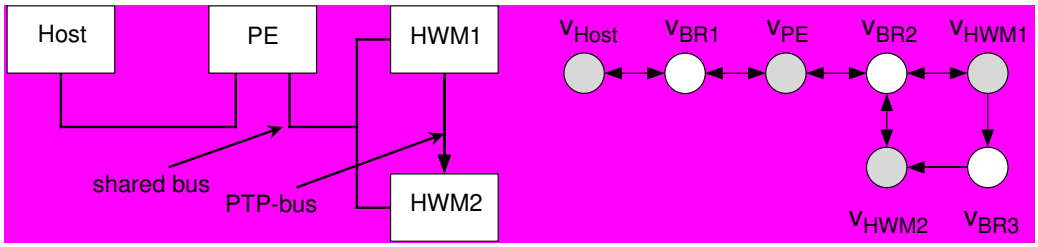


Figure 2.2 An architecture block diagram (left) and its abstract graph model G_A (right): a host machine or processor (v_{Host}) is connected to additional hardware with a bidirectional link (v_{BR1}). The additional hardware consists of a processing element (v_{PE}) and two hardware modules ($v_{HWM1/2}$) which are interconnected with a shared bus (v_{BR2}) and a point-to-point link (v_{BR3}).

With an abstract architecture graph the structural view of a system or a subsystem is described. Many properties of the architecture are still missing, e.g. the computational capabilities of a specific processing node, or the characteristic of a communication channel. With other words, for each node in the architecture graph G_A a behavioural model is needed.

2.2.4 Communication Model

The scalability of a communication algorithm in parallel environments generally depends on the communication volume, the available bandwidth, the buffering cost and the latency. A communication model should be realistic but simple enough to allow the algorithm designer testing his approaches on a wide range of machines and getting predictable performance estimation results. The *logP model* [37] seem to be a promising approach covering the mentioned issues. Initially developed for distributed memory parallel machines it may also be used for any kind of network topology which may be modelled with point-to-point connections. A modified logP model is presented such that long messages are modelled correctly as opposed to the original [37]. For simplicity, the modified model is also referred to as logP model.

For all nodes $v_i \in V_A$ of the architecture graph the logP *communication parameters* are:

$L_i = L(v_i)$ Upper bound on the latency: time to communicate one single word from the source processor to the memory of the destination processor. L_i is associated with communication nodes of the architecture model—$v_i \in V_A \setminus V_{PE}$—and in most cases will be a constant value. To model internal communication, the latency may also apply to computational nodes, thus $v_i \in V_A$.

$o_i = o(v_i)$ Overhead: time that a processor is occupied in the transmission or reception of data. During this time, the processor cannot perform other operations. This parameter is always determined by computational nodes $v_i \in V_{PE}$. Consequently, o_i occupies computational resources.

$g_i = g(v_i)$ Gap[1]: the gap per word for long messages, defined as the time per word for a long message consisting of w_i words. The reciprocal of g_i is the bandwidth available on this channel. The bandwidth is associated primarily to communication nodes $v_i \in V_A \setminus V_{PE}$. To model internal communication, g_i also applies to computational nodes $v_i \in V_A$.

The implementation of the logP model determines how data transfers are treated. Fig. 2.3 shows a basic logP point-to-point communication from v_{HOST} to v_{PE} (refer to fig. 2.2). As in most modern networks, where the network controller has direct memory access (DMA), a transmission of a message concerning the previous function may overlap the calculation of the current function (fig. 2.3a). With the abstract architecture model described before, the overlap is modelled automatically due to different node types for computation and communication. As fig. 2.3 illustrates, the overlap is easily disabled if required.

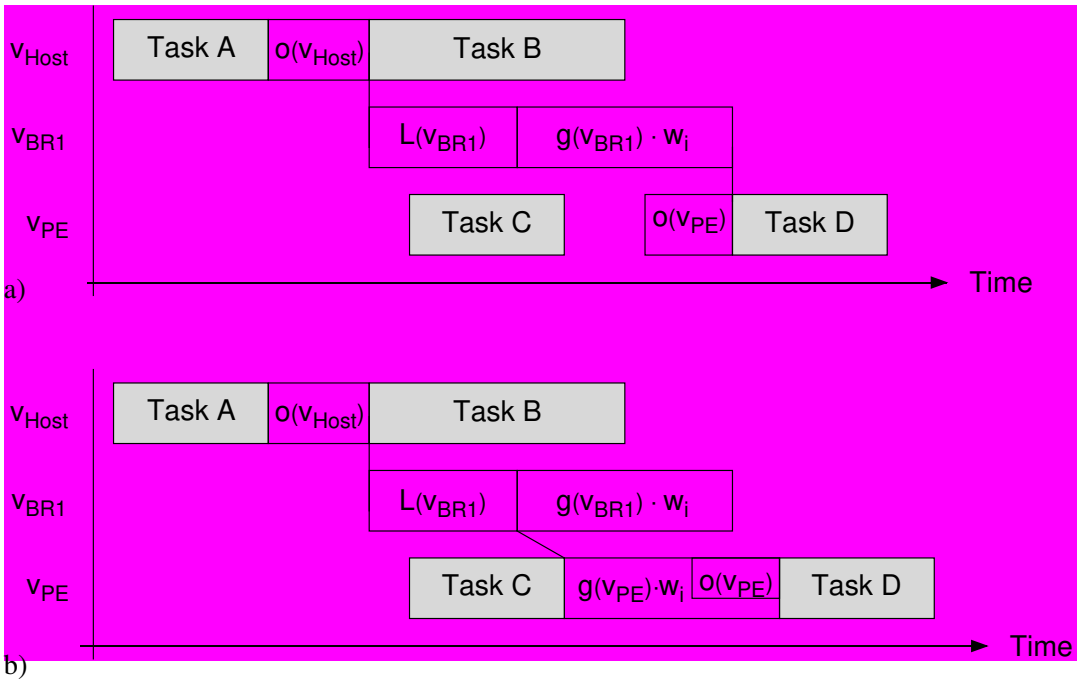


Figure 2.3 Modelling point-to-point communication with logP: task A sends a message consisting of w_i words to task D. The computational node v_{Host} is able to overlap communication and computation. Consider the two situations, where
a) node v_{PE} is also able to overlap communication and computation.
b) node v_{PE} may not overlap communication and computation.

1. It should be noticed that this definition of the gap g differentiates from the original: in [37], g is the minimum time between two consecutive writes or reads (of one word, or a small message) on a channel.

Since all computational nodes work asynchronously, the latency of a specific message is unpredictable but bound by the maximum value L_i, assuming that no error occurs. L_i applies each time a message of length w_i is communicated. Although the bandwidth $1/g_i$ usually is associated with communication nodes, in the following two cases the bandwidth additionally applies to computation nodes:

1. Overlap of computation and communication is not possible (e.g. disabled or non-existent DMA). In this case, a bandwidth $1/g_i$ is associated with $v_i \in V_{PE}$. If the message length w_i has reasonable size, $g_i w_i$ probably will dominate the overhead o_i (fig. 2.3b, v_{PE}).
2. Two successive tasks to be executed on the same computational node consequently are executed internal on the same node. Consequently, the communication between these tasks is composed of two times the overhead o_i, the latency L_i and the communication delay $g_i w_i$.

Computational nodes without the capability to overlap computation and communication usually obtain the same value for g_i as the connected communication channel. In practice, this is mostly the case for simple hardware devices, where the communication unit is coupled directly to the processor. In the opposite case, computational nodes are either idle or processing, but never occupied with communications (consider in fig. 2.3a the v_{PE} idle time between task C and the overhead o_i).

It is often possible to simplify the model by ignoring one or two parameters. For example, in algorithms that communicate small data packets infrequently, it is reasonable to ignore the bandwidth and capacity limits. In some algorithms, messages are sent in long streams which are pipelined through the network, so that message transmission time is dominated by the limited bandwidth, and the latency may be disregarded. The latency in most cases is a constant value, whereas situations may arise where the latency is given as a function of the message length. Then, for protocols supporting burst transfers, a different timing for short and long messages must be considered.

More complex communication schemes like broadcast or global summation (gathering of data: *reduce*) are mostly part of message passing libraries. Through cascading elementary point-to-point connections those collective communications may also be modelled with logP: first the message length w_i is reduced to one to obtain an accurate model. Then, for each collective communication primitive a communication schedule must be specified to obtain an accurate model in these cases. Of course this schedule depends on the computer architecture, the network topology, and mostly also on the network driver software. It is scarcely possible to overcome these dependences. Refer to [37] for example schedules. If the network has explicit hardware support for broadcast and reduce functions, models based on successive point-to-point communications will produce faulty estimations.

The modified logP model which was presented here can accurately predict communication performance with long messages of constant length. Many existing parallel machines have special support for long messages and achieve a much higher bandwidth for long messages compared to short messages (e.g. IBM SP-2 [53], Paragon, and most message passing libraries). This dependence could easily be included in the model by making the bandwidth a function of the message length. An alternative extension to the original logP model proposes a linear model for long messages, this way accurately modelling short and long messages, at the expense of an additional communication parameter [27].

To keep our model as simple as possible, a fixed message length per communication node w_i was assumed so that the latency and the bandwidth are independent of the message length. The resulting modified logP model as described above with the new definition of the parameter g_i has been used to model parallel architectures in section 2.2.8 and section 3.4.1, with the following restriction: broadcast and collective communications (reduce) are always modelled with successive point-to-point links, with messages of a constant length w_i. A different modelling technique which considers broadcast/reduce functions is introduced for GROMOS performance estimation on the IBM SP-2 (section 3.4.2). Furthermore it was assumed that the communication of the previous function may always overlap the computation of the current function.

2.2.5 Modelling Local Performance

To estimate the performance of a given parallel algorithm it is not enough to provide a model for interprocessor communication. Considering the system level architecture model of fig. 2.2, consisting of a host machine plus arbitrary processing elements, a behavioural model for computational nodes is still missing. Such a model must be universally applicable, independently of the target technology.

Due to the fact that the execution time of a function certainly depends on where the function is mapped to, for each computational node in the architecture a model for local computation performance is required: Perf(v_i), with $v_i \in V_{PE}$. This unit of measurement will be used later to determine the execution time of a task or a function. The model, which is presented here, does not take into account factors such as cache size and pipeline structures of processors, but differentiates the following two cases:

- The computational node is an application specific custom chip: an ASIC, a programmable gate array, or a comparable computation unit.
- The computational node is a general purpose processor: an embedded microprocessor, an entire embedded system, or even a workstation.

For the first case, the performance is usually known: it may be derived from the data sheet of the chip, or from the constraints in the requirements list of the behavioural model (e.g. if the chip does not yet exist). The unit of measurement Perf(v_i) for application specific chips is mostly the number of elementary operations per second. This is in conjunction with the specific function for which the proprietary processing element is designed. Depending on the application area of the custom chip, Perf(v_i) is sometimes given in units of MFLOPS or MIPS.

Assuming the latter case, where the computational node is a general purpose processor, the study of data sheets results in some peak performance information which is useless to predict the execution time of an application. This unsatisfactory situation was the initial motivation to develop standard CPU-benchmarks. For scientific applications SPEC benchmarks are widely used: SPECfp95 consists of ten programs with many floating point calculations, SPECint95 consists of eight programs with more likely integer operations. In principle, the SPEC performance is a *delivered* performance, but exclusively for one set of functions, namely the benchmark programs. Machine configurations and compiler options are well documented, and the source code of the benchmark programs is published. SPEC results are available as integer numbers for all workstations and PCs. These numbers denote the speed-up of the tested machine compared with a reference machine. The SPECx95 reference machine is a Sun SparcStation10 (40MHz).

Although it is possible to predict the execution time of a custom application through the use of SPEC results, it is preferable to measure the delivered performance directly at runtime. If an application (or function) specific delivered performance of the target machine is determined, $Perf_j(v_i)$ represents the performance for a given function v_j executed on the computational node v_i.

If the SPEC result is not available and direct measuring on the target node is not possible, the execution time must be estimated. This might be the case for specialised processors, e.g. for digital signal processors (DSP) or for embedded processors. A precise prediction is only possible if a detailed behavioural model of the function is available, combined with a thorough knowledge of the architecture of the processor.

Consequently, there are several possibilities to provide a performance model for computational nodes: runtime measurement, MIPS, FLOPS, SPEC ratios, etc. Finally, the choice of one specific unit of measurement is determined by the behavioural model of a function (table 2.1), such that the interplay of performance and behavioural models is ensured (section 2.2.8).

Several calculation schemes take an expanded set of parameters to determine enhanced system properties like overall cost, power consumption, chip area, and so on. The parallel computing model which was presented here is suitable for an analytical performance estimation of an algorithm. To estimate the performance of possible parallel GROMOS architectures, models for computational nodes are derived from results of direct runtime measurements (section 3.3.2). A FLOPS-based estimation was made in the case where the target node was not physically available (section 3.3.2). For system synthesis (section 2.3.2 and section 3.4.1 for GROMOS96P), the architecture model additionally captures the resource costs.

2.2.6 Algorithm Specification

The application specific graph-based model which is presented next consists of 1) *functional nodes*, 2) nodes representing control constructs, and 3) arcs representing either data flow or control flow.

Control/Data Flow Graph: CDFG

A control/data flow graph (CDFG, [15]) as sketched in fig. 2.4 is a heterogeneous or hybrid model combining the advantages of a control flow graph (CFG), for example a flowchart, with those of data flow graphs (DFG, [48]). A CDFG incorporates DFGs to represent data flow among activities or *operational nodes*, as well as a CFG to represent the sequencing of the DFGs. A CDFG explicitly shows both data dependences and the control sequence controlling the allocation of data flows. This is best shown by a piece of program code containing both computational and control elements which has been transformed into a CDFG representation (fig. 2.4). The transformation process typically assigns control constructs of the programming language to *control nodes* of the CFG, whereas assignment statements are mapped to operational nodes in the data flow.

The CFG shall have one start and one end node. Directed edges in the CFG do not indicate data dependence but determine possible execution sequences of the algorithm. Hence they do not carry data but only *tokens* without data type. Considering the example of an if–endif statement, an incoming token is led either to the outgoing edge of the triangle indicating the true-case (right branch in fig. 2.4) or to the edge for the false-case. Edges carrying a token are called *activated*

and enable subsequent functional nodes for execution. Assuming in our example ‘ldopair’ to be false, the ‘op1’ data flow starts executing. Thus, in addition to the rectangular boxes representing read and write from and to memory, three node types appear in fig. 2.4:

- operational or task nodes have one input and one output. They are ready to execute if all incoming edges are activated. After execution, the output or rather all outgoing edges are activated. Shadowed circles in fig. 2.4 represent operational nodes of the DFG and task nodes of the CFG.
- the conditional branch node is represented as a triangle with one input and two outputs. An associated boolean variable may not change once the control enters the CFG. Additionally, this boolean must not appear as a variable in the data flow graph. If all incoming edges are activated, the branch is determined according to the associated boolean variable: either all edges connected to the right branch indicating the true-case or all edges connected to the left branch indicating the false-case are activated.
- the corresponding join node is represented as an inverted triangle with two inputs and one output. The output is activated as soon as all edges of one input are activated.

The execution semantics of the DFG is very similar to the CFG: assuming that the ‘op1’ data flow starts execution, three data values (‘q’,‘r’, and ‘s’) are read from memory according to the specification (fig. 2.4). In addition to the activating token, data values are associated with edges. Operational nodes of the DFG execute if all incoming edges are activated. After execution, the result data is written to all outgoing edges, together with the activating tokens. When ‘op1’ reaches its terminating node, data item ‘a’ is written. The output of the task node ‘op1’ is activated, and the control flow continues with the end–if statement.

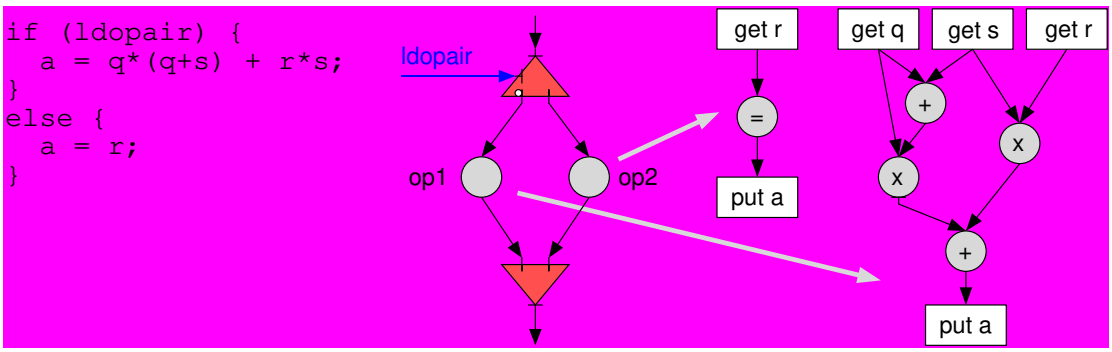


Figure 2.4 Sample control/data flow graph (CDFG): program code (left), control flow graph CFG (middle), and data flow graphs DFG (right).

With CDFG representations two types of parallelism may be specified. 1) As usual within the data flow: referring to the ‘op1’ data flow, the add and multiply operations may be executed in parallel. This is possible, because several tokens with associated data move downwards the graph, thus possibly enabling more than one operational node for execution. 2) Control flow parallelism is called task level concurrency and is also determined by the number of tokens. Unconditional fork and join nodes in the CFG to specify parallel tasks are not required. Following the execution

semantics as defined above a third task ‘op3’ in parallel with ‘op2’ is easily introduced in the CFG: with a second edge from the true-case output of the conditional branch node to ‘op3’, and from ‘op3’ to the right input of the node representing the end–if statement.

With a CDFG specification it is possible to represent any complex activities and control actions as it is required by the system. With the possibility of parallel control flow a CDFG specification goes far beyond the visualisation of a general purpose programming language. Moreover, several advantages qualify CDFG specification techniques against simple flow graphs:

- Compared with pure DFG or SDF (synchronous data flow graph, [48]) representations, a CDFG model combines both control domains. Parallel algorithms advantageously are specified with a CDFG: the parallelism is apparent, because both data and functional parallelism is effectively represented.
- Most algorithms—GROMOS’ MD algorithm included—are control and data flow driven and may be specified in an easy and straightforward way.
- Dedicated languages allow application specific CDFG models. On the one hand, the representation of control constructs and operational functions is not limited to the vocabulary of existing programming languages, and any complex activities are realised by adding new types of control nodes. On the other hand, an application specific CDFG may include only control node types which are really required to specify the considered algorithm.
- During design space exploration functional models are associated with nodes of the graph, the functional implementation is done later (gradual refinement, co–design technology)
- The CDFG description is independent of the partitioning into hardware and software.
- The specification is easy to change (replacement of functions, extension with new functions, introduce or reduce parallelism, etc.), which is also a design goal of GROMOS96P.

This flexibility induces some difficulties for tool implementors: despite the existence of a formal CDFG syntax specification (appendix C) extensive semantic checks are required to ensure consistent specifications. For example it would be easy to specify a syntactically correct data flow which is incompatible with its corresponding control flow—incompatible by means of deadlocks or data consistency errors. Other problems arise if a commercial tool should overtake a CDFG specification as input: compatibility issues would prevent user defined CDFGs. A summary of some different CDFG representations may be found in [16], and a dedicated CDFG specification model to describe the GROMOS parallel MD algorithm is introduced in section 2.4.

Dependence Graph Model

As it is the case for the parallel computing model, a CDFG is too complex for the high-level synthesis tool. Therefore, a simplified model is used on higher levels of abstraction integrating only the most critical design properties, or properties for which further optimisation is desirable.

The simplifying approach converts the CDFG into a simple DFG by choosing one specific control flow (e.g. the “worst case”), leaving away any control constructs. In addition, data dependences are replaced by explicit communication nodes. Although the execution semantics of the resulting graph is the same, it is not a DFG in the conventional sense and therefore it is referred to as *dependence graph*. The dependence graph $G_D(V_D, E_D)$ consists of a set of *functional nodes* and *communication nodes* $v_j \in V_D$ and edges $e_k \in E_D$. Functional nodes (grey)

typically are representatives for complex functions in a high level specification. Associated with functional and communication nodes (white) are execution times of executable implementations or appropriate models. The dependence graph model incorporates only one node type: whether a node is a communication or computation is just an attribute and has no influence on the execution. Similar to the abstract architecture model, edges of the dependence graph G_D model directed links between functional nodes and communication nodes.

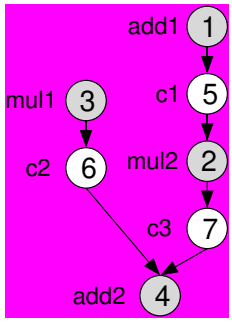


Figure 2.5 Sample dependence graph model $G_D(V_D, E_D)$, representing the 'op1' data flow of fig. 2.4.

To keep the example model simple and comprehensible, the functional nodes of the dependence graph (fig. 2.5) represent the elementary operations of the 'op1' data flow of fig. 2.4 (elementary operations instead of complex high level functions). With the dependence graph model, calculation and communication are separated in accordance with the abstract architecture graph. One important drawback of the dependence graph is that communication may not overlap calculation of the same function.

Analogous to the structural description consisting of the architecture graph plus associated communication and performance models, the dependence graph is completed with an appropriate behavioural function model.

2.2.7 Modelling of Functions

Analysing an existing implementation is easily done with standard profiling and analysis tools. The result is an exact information per function about execution time, memory demand, I/O, etc. on a specific machine. However, the goal of modelling is to provide a generic, machine independent functional model, whether or not a function is already implemented.

Functional nodes of the dependence graph represent parts of the parallel algorithm. In contrast to *communication parameters* (e.g. logP parameters) which are derived from experimentation or data sheet, *function parameters* depend on the algorithm and its *problem parameters* and are assigned to functional and communication nodes. Simple algorithms consists of one problem parameter n, for example a sorting algorithm with n elements to sort. More sophisticated applications incorporate several problem parameters, possibly even control variables. In practice, control parameters are not taken into account, assuming either a worst case or a typical control flow. In the following, $\mathbf{n} = (n_1, n_2, \ldots)$ represents the collective of all relevant problem parameters.

Function parameters specify the computational complexity of all functional nodes in the algorithm specification, as well as the local memory requirement and the amount of input and output data (table 2.1).

The knowledge of the underlying algorithm is essential to identify specific function parameters in practice. Only with this knowledge it is possible to model the function parameters with formulae in terms of **n.** If a function is already implemented, the employed algorithm is determined, and the function parameters are identified by a logical analysis of the algorithm. If a function without an appropriate implementation has to be modelled, one has to choose or develop an algorithm for this function. Once the algorithm is specified, all functional parameters according to table table 2.1 may be determined. The resulting functional model is validated by experimentation with different parameter sets **n** on possible target architectures.

O(**n**)	complexity of the function: the number of elementary operations
M(**n**)	Memory requirement [word]
B_{In}(**n**)	the amount of dynamic input data [word]
B_{Out}(**n**)	the amount of dynamic output data [word]

Table 2.1 Basic parameters of the functional model

2.2.8 Modelling the Algorithm–Architecture Relation

This step comprises the combination of the *dependence graph* (fig. 2.5) with the *architecture graph* (fig. 2.2, right). The resulting *specification graph* (fig. 2.6) has additional mapping edges relating the nodes of two neighbouring graphs.

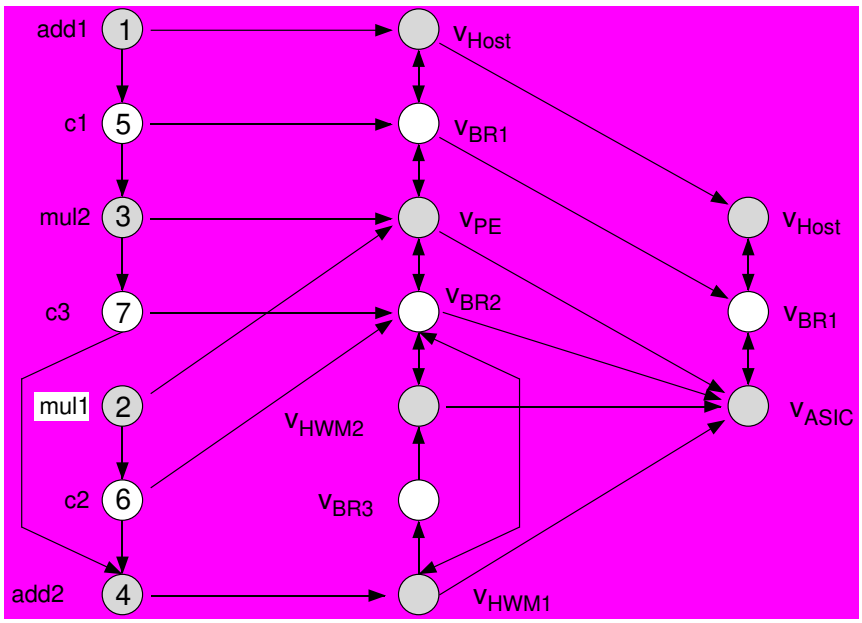


Figure 2.6 Mapping algorithmic functions (dependence graph G_D, left) to hardware resources (two architecture graphs G_A, middle and right).

A specification graph may consist of more than two architecture descriptions referring to different levels of abstraction. In fig. 2.6 on the right a system level description is added to the architecture graph. Note that it may be useful to map communication nodes to computational nodes of the architecture graph: if both predecessor and successor of a communication node are mapped to the same hardware resource, the communication is internal. Therefore, the communication parameters must also be available for computational nodes of the architecture specification (one can imagine that internal communication is not free).

The mapping edges describe one possible binding of algorithmic functions and communications to hardware resources and define a *feasible system* implementation. Algorithmic and architecture models are merged enabling analytical analysis of a specific implementation. The performance model as introduced in section 2.2.5 is well suited for an arbitrary function mapping: testing each function $v_j \in V_D$ separately on all possible processing nodes $v_i \in V_{PE}$ results in the function specific performance model $Perf_j(v_i)$. If a function v_j is mapped onto a computational node v_i, its execution time is given by

$$T_j^{comp}(v_i) = \frac{O_j(\mathbf{n})}{Perf_j(v_i)} + 2 \cdot o_i \tag{2.1}$$

Equation (2.1) applies the logP communication model with a constant message length w_i, assuming communications overlapping computations (fig. 2.3). The overhead o_i is independent of the communicated amount of data.

Most important is the interplay of the complexity $O_j(\mathbf{n})$ with the performance model $Perf_j(v_i)$: if the complexity model delivers the number of elementary operations, e.g. the number of force calculations, then the performance model must be scaled to the same unit. Considering the normal case where all possible target computational nodes are available, the function v_j is profiled on each target, and $Perf_j(v_i)$ is obtained applying (2.1). Remaining deviations are due to the abstracted or simplified functional model.

For the case that a processing node v_i is not available for testing, a standard CPU benchmark may be used instead (if the SPEC information of v_i is available): the function v_j is executed on a reference machine for which a SPEC benchmark is also available. Then, the execution time of v_j on v_i is estimated through the use of the ratio of the SPEC results. This “heave ho” method has been developed and legitimated in [45], where it has been shown that the inaccuracy lies around ±10% if the benchmark is chosen carefully. The inaccuracy is the deviation between the SPEC estimation and the measured execution time. The choice of the benchmark type depends on the characteristic of the function. As an example, a typical GROMOS function is scaled in section 3.3.2.

To calculate the execution time of communication nodes $v_j \in V_D$, at first the number of communicated messages B_j/w_i is determined. The latency L_i contributes to the overall communication time each time a message of length w_i is communicated (fig. 2.3).

$$T_j^{comm}(v_i) = \frac{B_j(\mathbf{n})}{w_i} \cdot (L_i + g_i w_i) \tag{2.2}$$

Once the mapping is fixed and the execution times are associated with mapping edges of the specification graph, the *schedule* for communication and computation may be determined. Different algorithms to accomplish this task are in common use, e.g. ASAP (as soon as possible),

ALAP (as late as possible), list scheduling, ILP (integer linear programming), and many more [16]. If a satisfactory schedule fulfilling the optimisation or design goal may not be found, the proposed implementation is not a *valid system*. Then, the designer may try to modify the system representation (architecture, algorithm) and their parameters (summarized in table 2.2), or try different allocations and bindings until a valid schedule may be developed. This iterative characteristic was the motivation to apply a semi-automated approach which is described in the next section.

Complex operations are represented by a functional node mapped on, for example, a dedicated hardware chip. Of course an exact prediction of cost, execution time and memory demand is not possible for such bindings, and an exact measurement implies the synthesis of these components. With the proposed modelling technique sufficiently good estimations are possible if computational nodes are available for testing.

Parameter	**Explanation**
$G_A(V_A,E_A)$	architecture graph with computational and communication nodes $v_i \in V_A$ and edges $e_l \in E_A$.
$G_D(V_D,E_D)$	dependence graph with functional and communication nodes $v_j \in V_D$ and edges $e_k \in E_D$.
P_{tot}, V_{PE}	total number P_{tot} of processing elements V_{PE}, $V_{PE} \subset V_A$, with $\lvert V_{PE} \rvert = P_{tot}$. []
L_i	latency of communication node $v_i \in V_A$. [s]
o_i	communication overhead of computational node $v_i \in V_{PE}$. If the communication is internal, o_i models the overhead of the context switch. [s]
g_i	gap: inverse of the communication bandwidth of a communication node $v_i \in V_A$. [s/word]
w_i	message size (number of words) for which the latency and bandwidth is scaled. [word]
$Perf_j(v_i)$	performance of function $v_j \in V_D$ when executed on computational node $v_i \in V_{PE}$. Delivers the number of elementary operation per second. [EOP/s]
$\mathbf{n}$	the collective of all problem parameters $\mathbf{n} = (n_1, n_2, \ldots)$. []
$O_j(\mathbf{n})$	complexity of functional node $v_j \in V_D$: the number of elementary operations. [EOP]
$M_j(\mathbf{n})$	memory requirement of a functional node $v_j \in V_D$. [word]
$B_j^{In}(\mathbf{n})$, $B_j^{Out}(\mathbf{n})$	the amount of dynamic input and output data of functional node $v_j \in V_D$. [word]
$T_j^{comp}(v_i)$	execution time of functional node $v_j \in V_D$ which is mapped onto $v_i \in V_{PE}$. [s]
$T_j^{comm}(v_i)$	execution time of communication node $v_j \in V_D$ which is mapped onto $v_i \in V_A$. [s]

Table 2.2 Model parameter glossary

2.3 Design Space Exploration

Commonly used as a generic term in a technological sense, design space exploration is mostly done at the start phase of system development. Considering an embedded system development, one can imagine an algorithm which may be realised as an ASIC, with a third party custom chip, or alternatively as software on the embedded controller. Exploring all these design alternatives, finding trade-offs, and finally coming to a design decision are typical design space exploration tasks. Successful design space exploration delivers an implementation concept. This concept is not proven to be optimal with regard to all considered parameters but provides a solution probably fulfilling the constraints. This cannot be guaranteed, because design decisions based on assumptions must not necessarily coincide with the reality. Independently of the system and the design goal, it seems that exploration always comprises similar tasks:

1. Working out a requirements list of the final product including constraints (e.g. cost, performance).
2. Perform a market analysis on existing approaches or solutions and check for in-house reusability.
3. Develop new approaches or concepts taking into account as many design alternatives as possible. Alternatives incorporate different hardware architectures, system and subsystems partitioning, algorithm modifications, etc.
4. Provide assessment functions to compare the design alternatives.
5. Make a preliminary design decision based on the results of the assessment functions and the requirements list.
6. Check the preliminary concept against the constraints if necessary. Review the temporary design decision in connection with point 2, and come to a final decision.

A new semi-automated design space exploration method has been developed which is applicable to system level [44] [7]. With this method it is possible to test many different hardware architectures and algorithms against cost, speed, computation time and other constraints within very short time. The method is based on solving the system-synthesis problem through the use of an optimiser which is based on an *Evolutionary Algorithm* (EA).

With appropriate models, the EA is capable to automate points 3, 4, and 5 from the list above. The remaining manual work is to exploit the model parameters. The remaining exploration tasks were adopted to the Molecular Dynamics project as follows: in section 1.5.2 the GROMOS96P main goal has been stated. An extensive market analysis and a study of existing MD hardware and software has been done and is reported in detail in [45]. In this thesis only the most important results are summarized in section 1.5.1 and section 3.1. To explore the GROMOS96P design space, the focus was twofold: 1) to find new approaches with system synthesis and conventional methods, and 2) to assess these approaches with performance estimation. Within this section, the method of synthesis based design space exploration is introduced with a simple exemplary system. Within chapter 3, the method is applied to find GROMOS96P architectures, and the performance of all implementation proposals is estimated and assessed.

2.3.1 Modelling for System Synthesis

As input specification, the system synthesis tool requires a specification graph as illustrated in fig. 2.7. This specification is very similar to the mapping of fig. 2.6 with the difference that between the dependence graph and the architecture graph additional mapping edges appear describing *all possible* bindings (dashed and solid lines). The insertion of mapping edges is one step requiring the designer's knowledge and strongly influences the behaviour of the synthesizer (performance *and* results). Similar to redundant mapping edges, the architecture graph is a superset of possible architectures with many alternative communication channels and computation nodes.

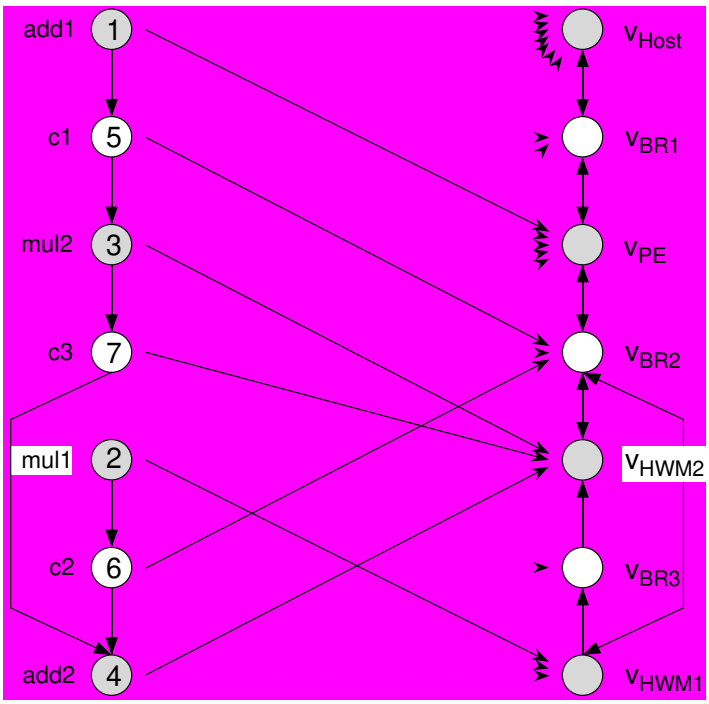


Figure 2.7 Specification graph, derived from the architecture graph G_A of fig. 2.2 and the dependence graph G_D of fig. 2.5.

A specification graph as in fig. 2.7 represents many possible implementations, all of which with different properties by means of the overall cost, performance and/or other parameters. The solid mapping edges indicate one possible binding of functions to resources, thus defining an implementation where the communication link v_{BR1} and the computational node v_{HOST} are not allocated.

Communication and functional model parameters are associated with mapping edges as described in previous sections: the execution time $T_j^{comp}(v_i)$ of a functional node $v_j \in V_D$, which is mapped onto the computational node $v_i \in V_{PE}$ of the architecture graph, is associated with the mapping edge between v_j and v_i. Analogously, the execution time $T_j^{comm}(v_i)$ of a communication node $v_j \in V_D$, which is mapped onto a architecture graph node $v_i \in V_A$, is associ-

ated with the mapping edge between v_j and v_i. Consequently, for each mapping edge in the specification graph the execution time of either a computational or a communication model must be available.

Since the synthesizer is not capable to process complex analytical execution time models, only integer numbers representing the respective time are allowed on the mapping edges. The same is true for the hardware cost of the architecture graph. Thus, as a precondition to use the synthesizer, execution times according to (2.1) and (2.2) must explicitly be determined for all possible bindings.

2.3.2 An Evolutionary Algorithm for System Synthesis

Evolutionary algorithms (EA) apply the principles of natural evolution to a random population. Natural evolution means selection and random variation through crossover and mutation, the population consisting of individuals represent implementations in the search space. Initially developed to solve global optimisation problems, the methodology has been adopted for system synthesis which is one of the key prerequisites of automated design space exploration [44]. The proposed methodology in [7] and [44] allows design space exploration during a simple optimisation run which is outlined in fig. 2.8. To start the optimiser, one has to specify the optimisation goal, e.g. latency minimisation (period) under resource constraints (cost).

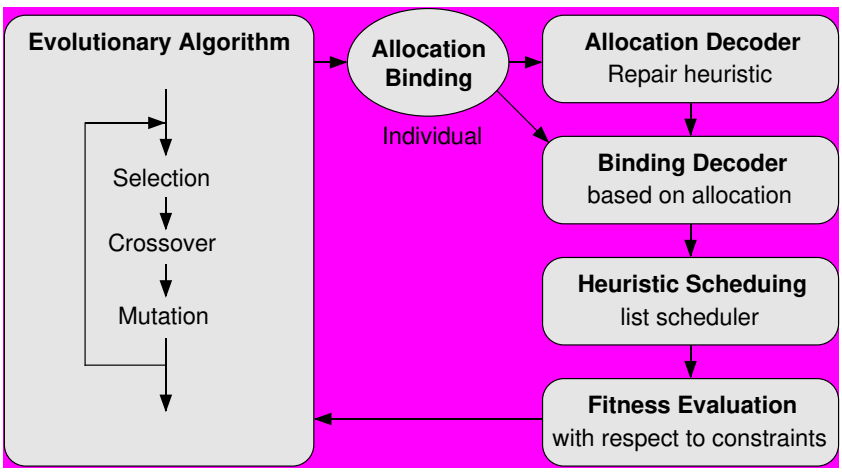


Figure 2.8 Application of an Evolutionary Algorithm (EA) to system synthesis.

Optimisation is an iterative task repeating the following steps:

1. The input for the first iteration is a randomly generated population of feasible bindings. A feasible binding is any association of functional nodes to hardware resources according to the specification graph.
2. With a heuristic list schedule algorithm appropriate schedules for the whole population is generated
3. A fitness evaluation function assesses all individuals of the population.

4. Within the EA, the population is resized according to a selection criterion. Selection is controlled by the optimisation goal (*Pareto*, *cost*, *period*: see below).
5. The initial population size is restored with crossover and mutation algorithms. Due to the randomly chosen allocations, many suggestions of the EA are not feasible.
6. Infeasible allocations are partially repaired using a heuristic. Exact repair methods would perform badly, and if no feasible binding may be found in reasonable time, the respective individual is deleted.
7. Go to step 2.

The result of step 2 is a set of implementations, illustrated as points in fig. 2.9.

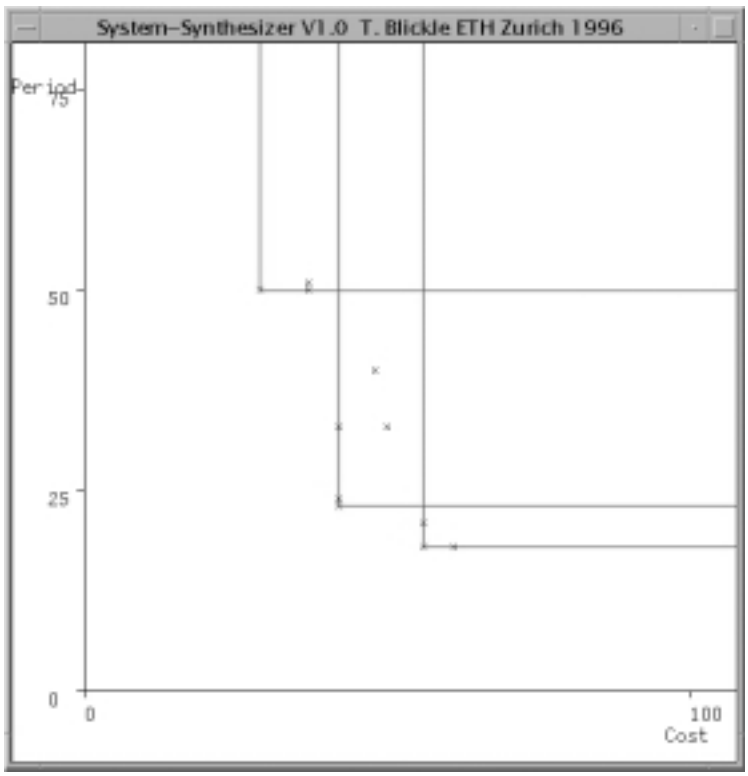


Figure 2.9 Design space and Pareto points

Depending on the selected optimisation goal, the EA tries to find the cheapest (*cost*) or the fastest solution (*period*), or it performs a *Pareto point* optimisation. Pareto points are indicated by lines in fig. 2.9. A point is denoted to be Pareto optimal if it is not dominated by any other point. Referring to the fastest two implementations of fig. 2.9, the cheaper of both is Pareto optimal because there is no cheaper implementation with at least equal performance.

When the scheduler is invoked, allocation and binding are fixed. Therefore, the scheduler must be capable to perform latency minimisation under resource constraints. Since most signal or image processing algorithms work on data streams, a scheduler supporting the iterative nature of

these algorithms would be desirable (pipelining, time stepping algorithms), since MD algorithms are iterative too. The list scheduling algorithm is only capable to schedule non-cyclic dependence graphs into non-iterative schedules.

Iterative schedules are characterised by different values for *latency* and *period*: the *latency* is the time which is needed to process the dependence graph one time. If the dependence graph is invoked repetitively, a result is delivered after each *period*. With a good pipelined schedule, *period* can be significantly smaller than the *latency* thereby improving the throughput. The basic list scheduler had been extended such that iterative algorithms are planned with minimised period. The basic idea of the iterative scheduler is to remove edges from the dependence graph such that the non-iterative list scheduler can be used. It should be noticed that the iterative scheduler is indeed implemented in the system synthesis tool [7]; unfortunately the vertical axis in fig. 2.9 is erroneously labelled with 'period' instead of 'latency', and tasks from preceding and succeeding steps do not appear in the Ganttchart of the schedule (fig. 2.10).

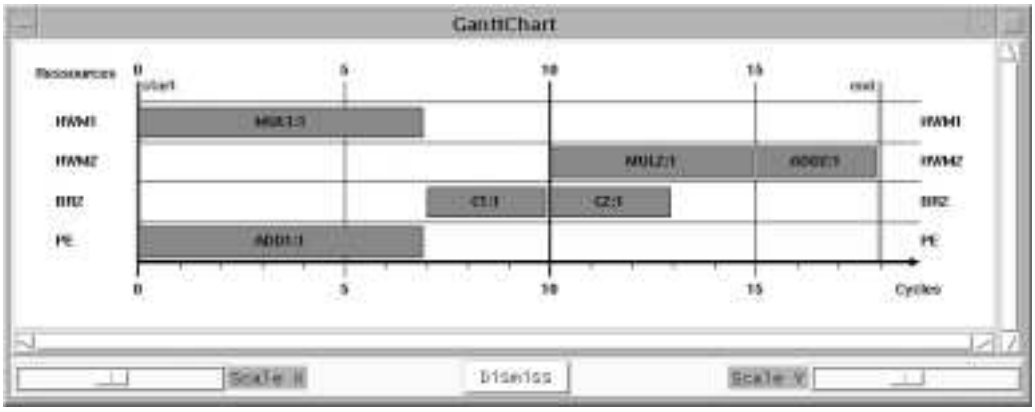


Figure 2.10 Schedule of an implementation. The depicted example refers to the Pareto point of fig. 2.9 representing the fastest implementation.

The input for the fitness evaluation function are—amongst others—the latencies of the implementations. Depending on the optimisation goal either Pareto points, very fast or very cheap solutions receive a high score. Implementations violating one or more constraints or the optimisation goal are penalised and receive a bad score.

2.3.3 Design Decision

Performing design space exploration with the synthesis approach solves the problem of mapping an algorithm, which is specified as a dependence graph G_D, onto a heterogeneous hardware/software architecture which is specified with an architecture graph G_A. This exploration technique takes into account: 1) the communication requirements, 2) finite computation and communication resources and 3) loop pipelining (iterative schedule). The method still shows some weaknesses:

- The models are very simple: only fixed computation and communication time is considered.

- Complexity functions and I/O demands of functional nodes are not taken into account dependence upon the problem parameters.
- Memory may not be modelled.
- Latency and overhead of the communication are not considered, and communication may not overlap computation of the same function.
- The EA delivers reasonable results only for large search spaces. Additionally, the EA is very sensible: marginal changes of one parameter delivers completely different results.

The method is very well suited if the search space has reasonable size. Then, the EA works as a filter fading out bad solutions while sometimes delivering surprising but promising proposals. One should always keep in mind that the EA is just one source of implementation proposals, and the designer's know how is essential to provide more variants. System partitioning and schedule for those additional architectures are determined manually or by using the EA too. To come to a final design decision, all suggestions are compared with each other. The comparison includes the assessment of all variants, based on estimations of the design quality.

To assess design alternatives, accurate performance estimation is important regardless of whether or not the design space is explored using synthesis techniques. To achieve a moderate accuracy, the estimation will probably also consider the control flow, combined with detailed functional models which describe the behaviour of the algorithm in dependence upon the problem parameters. The same is true for architectural models. They incorporate different properties like latency and overhead of communication channels, and dedicated performance models for computational nodes (section 2.2). With those models it is possible to predict the performance of an implementation with *varying* problem and architectural parameters. Together with the cost estimation, the choice for an implementation is straightforward.

Consequently, in most cases it will be impossible to come to a design decision without adding a level of detail to the models which have been used for synthesis. This is because the architectural differences between design alternatives may be small, even though a ranking list must be established. To explore the design space for the parallel GROMOS project (chapter 3), system synthesis was used [44], as well as performance estimations based on the models of section 2.2.

2.4 Software Synthesis

2.4.1 Overview

A short overview is given about the problem of synthesizing the software component of a system or subsystem. Related topics such as operating systems design, real time operating systems, and scheduling algorithms for operating systems [25] do not affect the parallelisation of MD algorithms and are not further discussed. The system for which software is to be generated shall be given as a behavioural description of the algorithm, preferably in the form of a graph model (e.g. in accordance with fig. 2.4). Functional nodes of the graph do not represent simple assignment statements but complex operations, for example the calculation of the pairlist. Due to significant differences between a graphical high level specification and abstractions at processor or instruction level (section 2.1.1), the task of software synthesis is performed in three steps [17]:

1. System or subsystem partitioning defines which functions belong to a program thread or *functional block*. Each program thread is a subgraph of the system level graph model. Depending on the mapping, individual threads are combined and compose a representation of the program to be executed on one processor.
2. From the graph of the program specification, high level source code (e.g. C code) for each processor is generated.
3. The compilation process translates the program into machine instructions by a software compiler and a processor specific assembler.

Structuring the system specification flow graph with threads has the advantage that parallelism is expressed by several ways: 1) data parallel characteristics appear within a function of a thread 2) functional parallelism within individual threads of a program and 3) functional parallelism between programs running on different processors. In many cases, a functional block is defined as a subgraph with a single entry node and a single exit node. Then, synchronisation or communication is at most necessary on thread level. This leads to a considerable simplification of the code generator.

Before generating code, one first has to decide if the schedule for each processor is determined at compile-time or at runtime. In the general case, the execution time of functions is not exactly predictable, hence called *non-deterministic*. This is caused by external events, internal exceptions or data dependent branches within a function. In this case, the operating system might be able to minimise processor idle times if the schedule is obtained dynamically (at runtime): since the operating system has the complete information about data dependences, a function is only executed if all required data is available. In the case of static or compile-time scheduling, the processor may idle because results of another task must be awaited, although an executable function would exist.

In data flow oriented systems it is often possible to predict the execution times very accurately, and a static schedule with minimum processor idle time may be determined. This is only possible if the execution times of all functions in the flow graph are made accessible to the code generator. Execution times of functions with internal control flow dependences are fixed assuming the worst case timing. The code generator takes the execution times into account to generate the scheduler. Control flows in the specification graph are included in the generated scheduler.

It is expected that off-the-shelf processors are delivered with appropriate compilers. Therefore, compilation and optimisation of source code into machine instructions is not further exploited.

2.4.2 Sequential Programming Environment

In the following it is assumed that a repetitive algorithm is given as a sequential program written in a high level language, e.g. FORTRAN or C. The GROMOS96 MD simulator would be a good example. As is the normal case for sequential programs, the execution sequence of (complex) functions is determined at compile-time. Within the iterations, an explicitly programmed *sequential scheduler* calls one function after the other, obeying the dynamic control flow of the algorithm. Referring to fig. 2.11, these function calls are identified with bidirectional arrows, and corresponding functions are depicted as striped white circles and labelled as e, f, g, h_*x*, and i. Some functions may occur as different versions (h_*x*), all of which implementing exactly the same part of the algorithm. The versions differ in that they are optimised for a specific target

machine. The function h is available as three versions h_1, h_2 and h_3 representing for example a standard sequential version (seq), a shared memory version (SMP) and a vectorized version (MPP), respectively. The source code consisting of all functions, the sequential scheduler, and the enclosing loop is translated into machine code with standard FORTRAN or C compilers. Complex makefiles control the compiler and linker to create programs targeted to different platforms. The linker selects the functions according to the target machine and determines whether additional proprietary libraries are required. This way, different platforms may be supported with dedicated optimised code, or the basic algorithm is implemented using special versions of a specific function.

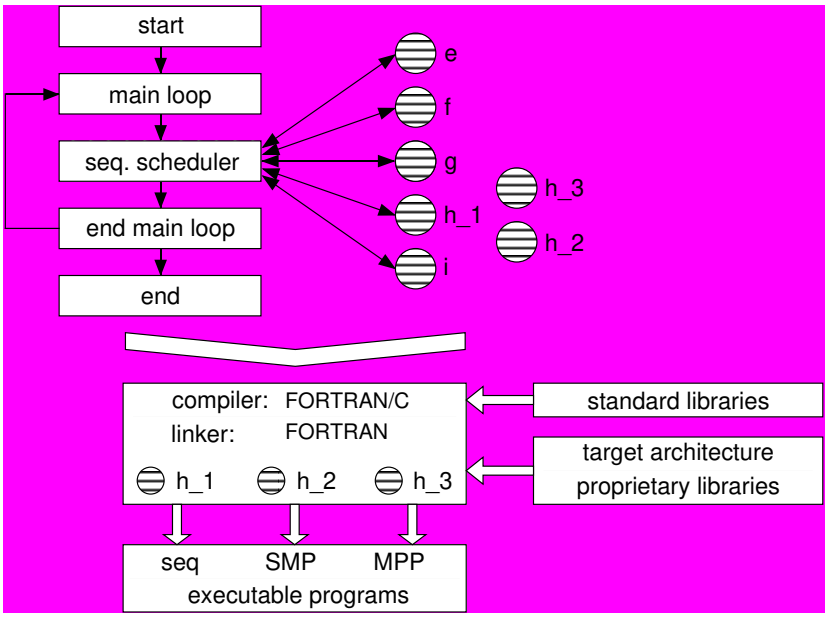


Figure 2.11 Sequential programming environment: outline of the GROMOS96 code organisation.

The described procedure to build executable programs applies regardless of whether different versions of functions represent parallel alternatives or just specialised sequential implementations. In other words, all functions in fig. 2.11 are treated the same. However, the situation drawn in fig. 2.11 matches the GROMOS96 code organisation, particularly multiple occurrences of the same functionality (section 1.5.2). If such a 'sequential' program is to be parallelised, two alternatives seem evident:

1. Adding new parallel functions as additional alternatives to possibly already existing multiple versions of functions. As a result, parallel executables would consist of a selection of parallel and sequential functions, and the selection of functions depending on the target architecture. This is the status quo of GROMOS96.

2. Writing one parallel program containing all versions of functions. This 'big' program specialises itself at runtime by choosing functions based on parameters.

The first concept was shown to be a bad solution (section 1.5), because it meets the requirements of GROMOS96P in now way. The second solution would result in unreadable code and slow executables. In addition, knowledge of the algorithm, the parallelisation strategy and parallel programming would be required of all users. This is in contradiction with the design goals. Therefore, an alternative intermediate approach has been pursued.

2.4.3 Proposed Parallel Programming Environment

The proposed concept (fig. 2.12, [41]) is especially well suited to situations where only parts of an existing implementation are subject to parallelisation. The parallelised part of the algorithm is preferably executed on distributed memory computational nodes using message passing communication based on MPI (Message Passing Interface, [21]), assuming that the same program is running on all nodes. Shadowed boxes in fig. 2.12 denote the handling of this parallelism.

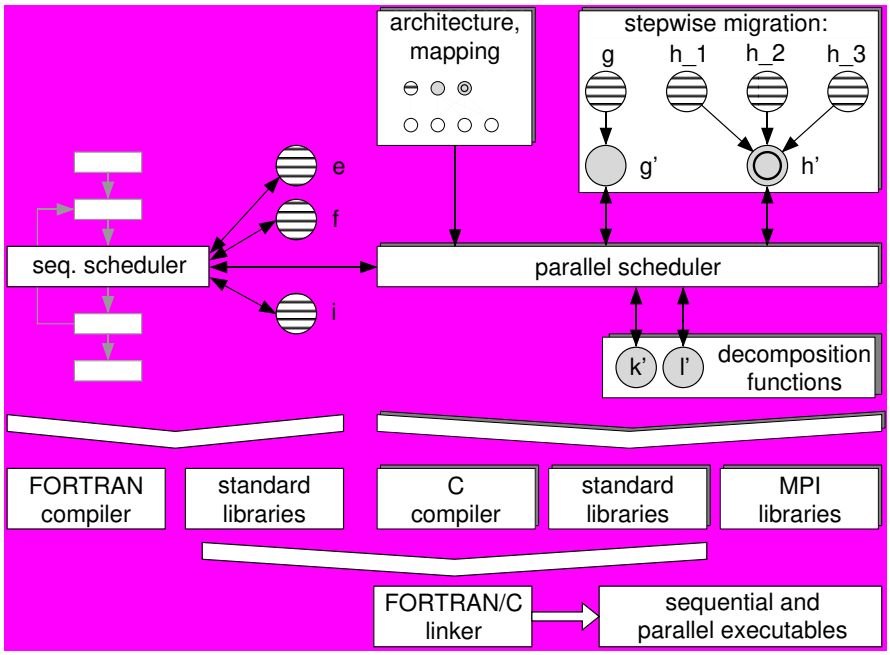


Figure 2.12 Overview of the proposed parallel programming concept.

In order to explain the concept, it is assumed that only one part of the algorithm must be parallelised, for example, the successive functions g and h (fig. 2.11). First, both subroutine calls are removed from the sequential scheduler (fig. 2.12). Instead of the function calls, an interface is inserted. The interface has one entry point and one exit point and represents the *parallel scheduler* function call. All remaining function calls in the *sequential scheduler* are left unchanged. Parallel and sequential schedulers differ in that the parallel scheduler is executed on all computational nodes, whereas the sequential scheduler may only run on the host.

Coming from an existing sequential program, functions that may contain internal parallelism (e.g. g and h_x) will be parameterized by following strict guidelines (“stepwise migration”, fig. 2.12). These guidelines are further explained in section 5.2.2 and affect mostly the formal specification of the calling interface of a parameterized function. To distinguish parameterized

functions by name, they are supplemented with an apostrophe. Parameterized functions comply with the rules of the chosen parallel decomposition and appear as grey circles. They may be executed on more than one processor. All parameterized functions are either of type parallel or sequential. A sequential parameterized function—for example g'—performs the same operation on the same data, whereas parallel parameterized functions—depicted with an inner circle: h'—perform the same operation on different sets of data. Thus, parallel parameterized functions exploit the domain decomposition.

The parameterization process removes all hardware dependences from algorithmic functions. Therefore, different versions of the same function, h_1, h_2, h_3, may be replaced by one parameterized version, h', which is valid for all possible target architectures. The hardware dependences are specified separately through the use of a simple architecture graph and the appropriate binding of algorithmic (parameterized) functions to computational nodes of the architecture graph ("architecture, mapping", fig. 2.12).

A domain expert adds special "decomposition functions" k' and l'. Decomposition functions are parameterized by definition, but never parallel, and determine the data set per processor on which parallel parameterized functions work. Consequently, g' and h' must be data parallel functions co-operating with decomposition functions.

2.4.4 Code Generation from Control/Data Flow Graphs

To find a suitable parallel implementation of an algorithm, the design space of possible solutions must be defined. If the design space is explored using synthesis methods (section 3.4), a system level flow graph model will be naturally available. In addition, functional models, system and subsystem partitioning, and appropriate schedules of possible parallel systems are also determined. The goal is to use this preliminary work for further *automated* refinement.

To accomplish automation, the approach as illustrated in fig. 2.12 is suggested. The *parallel scheduler* represents the parallel part of the algorithm. Similar to the *sequential scheduler*, parameterized functions are successively invoked by procedure calls, following the control flow rules. Consequently, the parallel scheduler is specified as a control/data flow graph (CDFG) very similar to that of fig. 2.4. This CDFG is a subgraph of the system specification graph and also has dedicated entry and exit points coinciding with the interface which has been inserted in the sequential scheduler. The CDFG contains not only parallel and sequential parameterized functions, but also a selection of decomposition functions. The mapping of these functions to computational nodes, as well as the graph itself, is specified with a domain-specific language.

With the intention to partially automate the parallelisation of the MD algorithm, a C code generator has been implemented [41][59]. The code generator is able to process the CDFG specification and to generate the parallel scheduler. A standard C compiler is used to translate the generated code. The remaining tasks are the same as for any sequential program: a FORTRAN or C linker is used to produce either sequential or parallel executable programs for different platforms. Additional communication libraries for interprocessor communication are possibly required.

The key issue of this approach is that sequential *and* parallel programs are derived from the same source code: a certain function called by the sequential scheduler is not allowed to coexist as parameterized function. The same rule applies for parameterized functions which are called by the generated parallel scheduler. This approach offers a number of benefits:

- Specification of the language with EBNF (Extended Backus-Naur Form, [65]) results in well documented and standardised interfaces. Additional semantic checks reduce the susceptibility to errors.
- Automatically produced source code is deadlock free. The fact that there is no unnecessary code leads to fast executables.
- Decomposition functions determine how the workload is distributed among computational nodes. First, the decomposition functions must be established. Then, new parallel functions are easily added.
- The concept is applicable to replicated data or data parallel algorithms, depending on the decomposition functions.
- Local parallelism is advantageously exploited if computational nodes are SMPs: 1) the data parallel characteristic is retained within the functions of a thread and may be exploited by the compiler 2) functional parallelism within individual threads (task level concurrency) is handled by the code generator.
- Design decisions are made based on a CAD supported specification graph, for example design space exploration [44]. Tasks like testing new algorithms or modifying algorithms by replacing functions, or generating different implementations for different architectures, are streamlined through the use of a dedicated code generator.
- Parallel and sequential program versions are derived from the same source code: maintenance cost is reduced drastically, with a small impact on the sequential performance.
- An existing program may be parallelised stepwise, by starting with the most time consuming parts; most of the existing code is left untouched.

The main aspects of the presented parallel programming environment (fig. 2.12) are as follows. Together with a code generator and a CDFG specification (chapter 5), the environment may handle *one* parallel scheduler, and the same program is executed on all nodes. A parallelised program consists of *one* sequential scheduler, which is executed always on the host, and the parallel scheduler. The parallel scheduler is invoked by the sequential scheduler and is running on *all* allocated computational nodes. Parallelism of multiple programs must be handled by the operating system scheduler.

CHAPTER 3

Design Space Exploration

In order to find the optimal hardware architecture to accelerate the GROMOS96 distribution, extensive design space exploration has been done. Design space exploration, as it is presented here, comprises the following tasks:

- Studying existing third party dedicated hardware approaches as well as other parallel Molecular Dynamics implementations.
- Analysis and benchmarking of the existing sequential GROMOS96 software to find performance bottlenecks.
- Exploring the design space through the use of a system synthesis tool based on an Evolutionary Algorithm (EA) to find possible hardware architectures.
- Estimation of performance and cost of implementation candidates. For that purpose, analytical models of the GROMOS algorithm and possible architectures are developed, applying the modelling techniques of chapter 2.

As a result of this chapter, it could be shown that a general purpose distributed memory multiprocessor or a workstation cluster can fulfil the basic requirements. Performance estimations have shown that with a high performance network running MPI (Message Passing Interface [21]) the delivered performance is satisfactory, provided that a spatial domain decomposition scheme is applied. Although for dedicated hardware or special purpose processors a better performance is predicted, the complexity of hardware and software components of these approaches contradicts important design goals such as extendability, platform independence, development cost and the overall ease of use of GROMOS96P.

3.1 Existing Approaches

The motivation to investigate existing or related approaches is twofold: Firstly it is important to verify if there is already a product on the market which may be used instead of a proprietary solution; secondly, if this is not the case, it is still possible to make much of a profit on third party experiences with already realised concepts. From the hardware point of view, two projects have recently been published, MD-Grape [40] and the MD Engine [63], both of which implementing dedicated hardware accelerators for classical Molecular Dynamics with ASICs as central force processors. Parallel implementations without specialised hardware support are AMBER [57],

CHARMM [34] and Gromacs [30]—three MD simulation packages competing with the current GROMOS distribution—as well as EulerGromos [36] and UHGromos [47], which are earlier parallel GROMOS implementations. These approaches had a direct impact on the development of GROMOS96P, some less related projects have also been investigated [43][45].

3.1.1 Specialised Hardware for Molecular Dynamics

The MD Engine card [63] consists of four ASICs to calculate forces and the Virial. Up to four MD Engine 6U-VME (Versa Module Europe) chassis can be connected to a host computer (SunUltra2) via a Sbus (SUN peripheral bus) interface, and up to 19 MD Engine cards can be placed in each chassis. Thus, the system is theoretically scalable up to a total of 304 processors, whereas the realised MD Engine consists of 76 processors. With 76 processors, the MD Engine reaches a speed-up of 48 compared with a host only simulation of 13,258 atoms. The force processor has a hardware pipelined architecture to calculate the total nonbonded force, the Virial is calculated in the same manner as it is the case for the GROMOS algorithm. The major difference is how Coulombic force is treated. GROMOS follows the charge group concept and applies the same cutoff criterion for Coulombic forces as for the Lennard-Jones potential. The MD Engine does not apply a cutoff criterion to Coulombic forces, thus calculates all particle pairs. Nevertheless, a pairlist is required because a cutoff criterion is applied for the Lennard-Jones interaction calculation. During the scan of all particle pairs for Coulombic forces calculation, the pairlist is easily constructed. Consequently, the performance behaviour of the MD Engine is worse than a GROMOS simulation, even if the pairlist would be reconstructed within each iteration using the brute force algorithm.

The GROMOS force field incorporates the Coulombic dipole-dipole interaction and the corresponding reaction field correction in the Lennard-Jones potential. Yet, the resulting equations for the GROMOS nonbonded interaction (equation (1.26), [63]) are similar to the equations which are implemented in the MD Engine hardware. Therefore, the ability to realise an appropriate ASIC considering the GROMOS force field is not out of scope. In addition, it may be assumed that such a dedicated GROMOS force processor reaches at least the same performance as the MD Engine: to calculate one nonbonded interaction within 400ns [63]. Despite of the fact that the MD Engine has hardware support for very important features like pairwise forces and periodic boundary conditions with a cubic box, promising concepts concerning the pairlist construction are missing.

Similar to the MD Engine, one MD-Grape board (6U-VME) consists of four MD chips. The MD chips are also capable to handle an arbitrary (that is, not only $1/r^2$ dependent) central force. Hardware pipelines allow the calculation of one interaction per clock cycle if the direct summation algorithm is used. Direct summation means that the interaction of all particle pairs is calculated without using a cutoff criterion. Then, MD-Grape reaches the peak performance of 4.2 GFLOPS per board.

Therefore, the principal application of MD-Grape is cosmological N-body simulation because it requires a force without cutoff. If a cutoff is applied, MD-Grape stores the particle coordinates in the local memory in a way that the cell index method (section 4.1.1) is supported and only particles residing in one of the 27 neighbouring cells are fed to the force processor. A pairlist is not calculated explicitly, the pair separations of all particle pairs reaching the force processor is calculated. If the distance is greater than the cutoff, the calculated force is rejected. This leads to a much worse performance compared to the direct summation, because a considerable amount of

calculated interactions are not used, and the gain of deep hardware pipelines is significantly reduced. Moreover, the address processor provides the particles of all 27 neighbouring cells, so pairwise identical forces are calculated twice. In addition, if the Virial is to be calculated, the main loop is executed twice. As a consequence, if the Virial is to be calculated and a cutoff is applied, an overhead of a factor of four occurs. The MD-Grape hardware is limited to 43,690 particles. If there are more particles in the system, an additional loop over the particles must be executed. The most interesting concepts of MD-Grape are the hardware address unit which supports the cell index method and, again, the hardware pipelines to calculate the forces.

Both hardware approaches use the replicated data approach: data distribution is simply done with a broadcast to all boards, since each board stores all coordinates of the system. Both ASIC solutions are able to calculate the basic Lennard-Jones interaction potential (1.9), either combined with the Virial calculation, or the Coulombic force, but not both. Still missing is a concept how more sophisticated features like solute interactions, or perturbation (section 1.4.3), is to be incorporated in an ASIC-based coprocessor architecture.

The Grape-6 [52] specialised hardware is aimed at large stellar systems to simulate the evolution of star clusters with up to one million stars. To achieve the projected peak performance of 200 TFLOPS, the total number of 4096 pipeline chips is required. Six pipelines per chip are especially designed to calculate the gravitational force between particles, the integration step is done on the host. Two pipeline chips are packaged with four fast SSRAM (Synchronous Static Random Access Memory) chips in a multi-chip module (MCM). The MCM internal data transfer speed is around 1.2GByte per second. 2048 MCMs are organized into 16 clusters each with eight processor boards. The particle coordinates are replicated on the off-chip memory of each of the 16 clusters. To divide up the workload, for each cluster a disjunctive set of particles is grouped together and stored in the MCM on-chip memories. Thus, within one cluster, different chips calculate the forces of the same set of particles, but from different particles. Partial forces only occur within a cluster and must be summed up through a reduction network. Given a system consisting of one million particles, each pipeline processor has to calculate more than 100 million interactions, since interactions are always calculated for all possible particle pairs. Generally, good scalability can be achieved with a large number of interactions per processor and time step. The usage of MCMs and the employed domain decomposition are two promising concepts, both of which applicable to the GROMOS96P project.

The Grape-6 technique applied to Molecular Dynamics would mean calculating all particle pairs highly efficient (pipelines). On the other hand, adopting a pairlist method would reduce the number of considered interactions. The MD Engine and MD-Grape hardware approach the tradeoff between the two extremes. Generally, for MD systems of reasonable size, the gain of pairlist methods dominates. Because the usage of a pairlist significantly reduces the number of interaction calculations per processing node, the communication time will become more significant.

3.1.2 Parallel Software Implementations

AMBER and CHARMM software packages both have been shortly discussed and assessed in section 1.5.1. The initial version of Gromacs [30] was a specialised parallel computer for MD simulations, consisting of 32 commercially available i860 boards with two 8 Bit wide parallel ports to interconnect the boards in a ring. After the hardware became outdated, Gromacs has been further developed for general purpose parallel computers. Currently available versions support

shared memory machines and message passing systems based on MPI or PVM (Parallel Virtual Machine). In contrast to EulerGromos [36] and UHGromos [47] (section 1.5.2), Gromacs [30] incorporates a parallel twin-range pairlist method. In addition, an *atomic Virial* calculation is employed, in this way enabling the implementation of the efficient domain decomposition scheme (section 4.2). Nevertheless, none of the three GROMOS derivatives support octahedron periodic boundaries in their parallel versions.

Differently to dedicated hardware approaches, the gain of studying parallel software implementations of other MD simulators is lowered. Various parallelisation techniques are well known and described in section 4.2, where different decomposition schemes with their advantages and disadvantages are compared. To predict the performance of possible GROMOS96P solutions, two different schemes are applied to several architectures in section 3.4.2. Parallel software implementations differ mainly in the number of supported parallel functions, e.g. parallel twin-range. The main advantage of pure software solutions is that computational nodes consist of general purpose processors. Therefore, it is possible to parallelise the whole algorithm and not only the nonbonded interaction calculation. Parallel programming is simplified significantly if standard communication libraries like MPI or PVM are used.

3.2 GROMOS96 Analysis

As a foundation for all subsequent work, GROMOS96 has been analysed thoroughly to find performance bottlenecks and entry points for optimisation. The performance analysis is based on simulations of typical molecular systems, which are defined as performance benchmarks in section 3.2.1. Running the benchmarks with GROMOS96 delivered profiling results which are documented in [45] for many different platforms; section 3.2.2 summarizes profiling results with respect to the reference platform (SunUltra30). Properties of the reference machine are specified in appendix B. Analysing the execution time of standard benchmarks allows the calculation of best-case speed-ups for different grades of parallelism (section 3.2.3). Best-case speed-ups reveal those parts of the algorithm which *must* be parallelised in order to reach the performance goals.

Models of hardware architectures and the MD algorithm are developed in section 3.3. Accurate models are required for design space exploration, especially for the tasks comprising system synthesis and performance estimation. An enhanced version of the logP parallel computation model is used to model parallel architectures. By using this architecture model together with analytical models of GROMOS96 functions, the execution time of computation and communication may be predicted (in conjunction with profiling results). The specification of the algorithm as a dependence graph serves as input for the system synthesizer.

In order to invent as many target architecture proposals as possible, the knowledge about existing parallel MD approaches was essential. Additional promising implementation proposals are expected to be found through the use of a system synthesis tool (section 3.4.1). To take a decision concerning the GROMOS96P parallelisation strategy, generic models of GROMOS functions and promising architectures were implemented in Mathematica [24] to estimate the performance of hypothetical GROMOS96P implementations (section 3.4.2).

3.2.1 Benchmark Specification

Two typical benchmarks are used throughout this thesis for performance testing. A typical simulation consists of one or more solute molecules, typically proteins, surrounded by a solvent, e.g. liquid water, with periodic boundary conditions applied. Both performance benchmarks consist of the same solute molecule (Thrombin) and the same solvent (water). The only difference is that the simulation box of one benchmark is a truncated octahedron (*Thr2*), whereas the other is a rectangular box (*ThrG2*). As an option, Thrombin may be simulated without water (*Thr1*), viz in a vacuum without periodic boundaries.

Measured times are always in units of seconds and refer to 100 simulation steps unless otherwise noted. Example simulations are always performed in three spatial dimensions, with double precision floating point arithmetic. The pairlist is updated every five steps, and the twin-range pairlist option is disabled. The SHAKE algorithm is used to handle constraints and the Virial is calculated. For a complete description of the benchmark systems, we refer to appendix A.

3.2.2 Profiling

Before developing hardware or parallel software, possible performance bottlenecks of the existing sequential software must be pointed out. To accomplish this, operating systems provide utilities for monitoring program execution and system activity. To trace the timing behaviour of GROMOS96, the unix profiling program *gprof* has been used. *gprof* produces an execution profile of a program. Parts of GROMOS96 profile data is summarized in table 3.1.

The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file which is created by programs compiled with the -pg option. Quantizing errors may occur if a function has a very small 'self' time. The granularity of the sampling is shown, but remains statistical at best. It is assumed that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called.

A SunUltra30 was used as a reference machine. The profiled MD simulation program *promd.64* is based on GROMOS96 FORTRAN source code with the exception that some timing functions are written in C. The number suffix (.64) indicates double precision floating point calculation. The reference machine and compilation details are also described in appendix B. The example listing in table 3.1 summarizes total execution times and call counts of the most time consuming functions of *promd.64*, sorted by decreasing time.

The majority of time (75.2%) is spent for the nonbonded forces calculation in subroutine *nonbml_*. Trailing underscores indicate FORTRAN functions, leading underscores denote library functions. Underscores are omitted in the following. Subroutine *nbnone* calculates the GROMOS96 pairlist without twin-range interactions, *shake* handles constraints, and *mdleap* is the MD-integrator. If the Virial is to be calculated (1.20), subroutine *prpvir* is called each time step, delivering the atom positions relative to the centres of mass of charge groups (equations (1.17) and (1.18)). Subroutine *clcgeo* calculates the charge group positions. The function *clcgeo* is called from *nbnone* because charge group positions are used only for pairlist construction. The function *force* is called from within the main simulation loop (main MD loop), which is part of *runmd*. The reason why *runmd* and *force* have a very small self time per call is because they consist of subroutine calls rather than performing local calculations. Indeed *force* comprises all func-

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 3074.56
seconds
   %  cumulative     self              self      total
 time    seconds  seconds    calls  ms/call    ms/call   name
 75.2    2312.39  2312.39      100 23123.90   23128.33   nonbml_ [7]
 20.5    2941.39   629.00       20 31450.13   31450.13   nbnone_ [9]
  2.4    3016.20    74.81   710003     0.11       0.11   _write [11]
  0.8    3040.39    24.19      200   120.93     120.93   shake_ [16]
  0.3    3048.27     7.88   705344     0.01       0.01   _lseek [17]
  0.1    3052.19     3.92      100    39.20     281.05   mdleap_ [15]
  0.1    3055.17     2.99  1096204     0.00       0.00   cenmas_ [20]
  0.1    3057.29     2.12                                 _mcount (985)
  0.0    3058.70     1.41      100    14.10      49.45   prpvir_ [18]
  0.0    3060.11     1.41      400     3.53       4.22   dihang_ [23]
  0.0    3061.28     1.17      200     5.85       5.85   angle_ [26]
  0.0    3062.44     1.16                                 oldarc [28]
  0.0    3064.95     0.48      100     4.80   29457.81   force_ [6]
  0.0    3065.78     0.41                                 fseek [12]
  0.0    3066.15     0.37      100     3.70       3.70   shiag_ [40]
  0.0    3066.83     0.34  1411892     0.00       0.05   _xflsbuf [10]
  0.0    3067.17     0.34                                 _fconvert [19]
  0.0    3068.05     0.26  1080500     0.00       0.00   __anint [51]
  0.0    3068.53     0.23        8    28.75      28.75   _setbufend [54]
  0.0    3068.74     0.21      397     0.53       0.53   _read [57]
  0.0    3069.15     0.20   539909     0.00       0.00   _memcpy [59]
  0.0    3069.89     0.18        1   180.00     180.00   .mul [63]
  0.0    3070.22     0.16   705340     0.00       0.05   _fflush_u [14]
  0.0    3070.99     0.15    35961     0.00       0.01   chpnre_ [55]
  0.0    3071.13     0.14   705325     0.00       0.00   _wrtchk [41]
  0.0    3071.41     0.14   137492     0.00       0.00   random_ [70]
  0.0    3071.69     0.14      100     1.40       5.50   conat_ [34]
  0.0    3071.82     0.13       20     6.50       6.50   clcgeo_ [75]
  0.0    3072.07     0.12     1442     0.08       0.08   _times [77]
  0.0    3073.48     0.06        1    60.00 2979995.82   runmd_ [5]
```

Table 3.1 *gprof* output of *ThrG2* benchmark: pairlist every five steps, 1.4nm cutoff without twin-range. Machine: SunUltra30.

tion calls for bonded (e.g. *cenmas*, *dihang*, *angle*) and nonbonded forces calculation as well as the pairlist construction. The rest of the MD algorithm (*mdleap*, *shake*, etc.) is calculated within the main MD loop. Before entering the main MD loop, initialization functions and functions to handle the preconditions, as described in section 1.4.2, are called from *runmd*.

Most of the simulation time is spent to construct the pairlist and to calculate the nonbonded forces. Except *shake*, all other tasks do not contribute anything worth mentioning. The library function *times* is called from software timers which were added to the GROMOS96 distribution primary to test *gprof* results. These timers provide similar features like a stop watch (start-stop, lap, reset) and are available anywhere in the code. It is therefore possible to decompose the non-bonded forces execution time into two parts: one part containing interactions involving solute and solvent molecules (solute-solute and solute-solvent interactions), and another part containing only solvent-solvent nonbonded interactions. Table 3.2 summarizes execution times of *promd* functions similar to the output of *gprof*, but by using the mentioned timers exclusively. Thus, the C-times delivered by gprof and the software timer may be cross-checked (table 3.2 and table 3.1).

Function	***Thr2* [s]**	***ThrG2* [s]**	***Thr1* [s]**
main MD loop	1,565	2,976	117.68
all forces (*force*)	1,540	2,928	114.16
integration (*mdleap*)	14	27	2.09
pairlist (*nbnone*, *clcgeo*)	228	607	3.94
all nonbonded forces (*nonbml*)	1,308	2,318	106.93
solute-solute and solute-solvent	365	366	106.90
solvent-solvent	943	1,952	0

Table 3.2 Benchmark execution times of 100 simulation steps, measured with C-timers, on the SunUltra30.

3.2.3 Reachable Speed-ups

Amdahl's law defines the speed-up as

$$\text{Speed-up} = \frac{T_{seq} + T_{par}}{T_{seq} + T_{par}/P_{tot}} \tag{3.1}$$

with T_{seq} the execution time of the sequential part of a program and T_{par} the execution time of the parallelised part of a program running with P_{tot} processors.

If P_{tot} is set to infinity, the result is a best-case speed-up for a given degree of parallelism. In table 3.3, best-case speed-ups for both *Thr2* and *ThrG2* benchmarks are listed.

Pl-Update/Cutoff ***Benchmark***	**Solvent–Solvent**	**Solvent–Solvent and Pairlist**	**Nonbonded and Pairlist**
5/1.4 *thrG2*	2.9	7.1	58
5/0.8 *thrG2*	1.5	9.9	21
10/1.4 *thrG2*	3.7	6.4	52
10/0.8 *thrG2*	1.9	7.2	50
5/1.4 *thr2*	2.5	4.0	54
5/0.8 *thr2*	1.6	5.7	17
10/1.4 *thr2*	2.9	3.7	52
10/0.8 *thr2*	1.9	4.3	13

Table 3.3 Maximum speed-ups for *Thr2* and *ThrG2* benchmarks for different degrees of parallelism.

The leftmost column describes the simulation parameter set of the examined MD simulations. The parameters are: pairlist update ratio, cutoff radius and benchmark type. The next columns refer to different degrees of parallelism as follows: 1) only the solvent-solvent part of the nonbonded interactions contribute to T_{par} 2) in addition to that, the pairlist is also parallelised 3) all

nonbonded calculations and the pairlist construction contribute to T_{par}. Results of the first and fifth row are derived from execution times of table 3.2.

In order to reach a speed-up of at least ten on realistic conditions (the *delivered* performance of the final implementation) the solvent *and* solute part of the nonbonded forces as well as the pairlist construction must be parallelised. The separation of the nonbonded interaction into solvent and solute is essential: the force field is significantly simpler for solvent-solvent interactions (section 1.4.3). The separation allows different functional models representing different implementations for the solute and solvent nonbonded part, respectively. In the following discussion, solute-solute and solute-solvent nonbonded interactions are abbreviated as *soluForce*, and solvent-solvent nonbonded interactions as *solvSolv*.

3.2.4 Function Models

According to section 2.2.7, the complexity, memory and I/O demands of the GROMOS functions are modelled. The function models as they are presented here incorporate GROMOS specific properties and parameters as follows:

- All important control flow alternatives of the MD algorithm are considered: octahedron or rectangular periodic boundaries, the pairlist update ratio, and the Virial calculation option. Most other flow controls (e.g. options for covalent bond forces calculation) do not contribute perceptibly to the overall performance. Perturbation would contribute, but is *not* modelled in the solute nonbonded interaction function. For simplification, the oblique simulation box is treated like a rectangular one.
- Models for complexity, memory and I/O demand are generic, i.e. all parameters as listed in table 3.4 are considered.
- Complexity models for the pairlist construction and *solvSolv* offer some finer granularity. The elementary operations (e.g. distance calculation) are further examined: the number and kind of floating point (fp) operations per elementary operation is established dependent on the problem parameters.
- The I/O model takes into account only dynamic data. Dynamic data change at least once during the simulation run.

A generic model for elementary operations delivering the number of floating point operations is very useful to predict the delivered performance of ASICs or processors with specialised or multiple floating point units [2]. During design space exploration, computational nodes consisting of specialised processors are often not yet synthesized and therefore not available for testing. Planned to execute exclusively one dedicated function, the internal architecture of the processor and the embedding environment is well known. The software part of the function which is implemented on this node probably is manually optimised or programmed directly with an assembler. On these conditions, the behaviour of a computational node is predictable, allowing the execution time of a particular function to be estimated on that node.

It is not recommended to apply the floating point model to general purpose processors or microcomputers, because that model would not lead to more accurate performance estimations. The reason is the execution behaviour of these computational nodes which is highly influenced by the cache hit ratio, the operating system, and other tasks running on the same node. Even

though, the floating point model facilitates the assessment of the quality of an implementation. The delivered floating point performance may be compared with peak performance rates of the processing node. The result of that comparison determines the maximal reachable performance gain of further optimisation.

Parameter	Explanation
ndim	number of dimensions to calculate (three or four)
nvcag	total number of solvent charge groups
nram	number of atoms per solvent charge group
npm	total number of solute molecules
ncag	number of charge groups in one solute molecule
nrp	the number of atoms per solute molecule
nragt	total number of charge groups
nattot	total number of atoms
r_c	the cutoff radius
V_{box}	volume of the simulation box
l4d	false if ndim equals three; true if ndim equals four
ldovir	true if the Virial is to be calculated
ldopair	true if a pairlist is to be constructed within the current time step
p1	the probability that due to the nearest image convention periodic boundary correction are applied for one component of the connecting vector between two particles (rectangular and octahedron box)
p2	only for octahedron boundaries: the probability that an additional boundary correction term is applied

Table 3.4 GROMOS MD simulation parameters and control variables. Generic models of GROMOS functions are based on these parameters.

Pairlist

The pairlist construction consists of two tasks: the determination of the particle positions and the neighbour searching itself. The position $\mathbf{R}_\alpha^{cg}$ of a solute charge group $\alpha = 1, \ldots, npm$ is taken to be its centre of the geometry (superscript 'cg'):

$$\mathbf{R}_\alpha^{cg} = \sum_{i=1}^{N_\alpha} \frac{\mathbf{r}_{i\alpha}}{N_\alpha} \tag{3.2}$$

The double index $i\alpha$ denotes atom i of solute molecule α which consists of N_α atoms. The position of a solvent charge group is taken to be the position of the first atom of the solvent molecule.

Distance calculations between charge groups are required for neighbour searching and are performed with periodic boundaries applied. The number of floating point operations for both tasks—position and distance calculation—are given in table 3.5.

Given the computational cost of the basic pairlist operations in table 3.5, we are mainly interested in the complexity of the algorithm itself, that is how many times these basic operations are executed. The centre of solute charge groups is calculated $\text{npm} \cdot \text{ncag}$ times (function *clcgeo*), which goes linear with the number of particles. Therefore, the algorithm is supposed to be optimal and not subject to optimisation. The solvent part is disregarded because nothing is calculated.

$$O(\text{clcgeo}) = \text{npm} \cdot \text{ncag} \tag{3.3}$$

$$B_{\text{In}}(\text{clcgeo}) = \text{ndim} \cdot \text{nattot} \tag{3.4}$$

$$B_{\text{Out}}(\text{clcgeo}) = \text{ndim} \cdot \text{nragt} \tag{3.5}$$

Operation	**Centre of Geometry**	**Distance: *ThrG2***	**Distance: *Thr2***

`+/-`	`(nrp/ncag-1)*ndim`	`(ndim)*2`
		`+ (ndim*p1)*1`
			`(ndim)*5`
			`+ (ndim*p1)*1`
			`+ ((ndim+1)*p2)*1`

`/`	`ndim`	`0`	`0`
`*`	`0`	`(ndim)*1`	`(ndim)*1` `+ (p2)*1`

Table 3.5 The number of floating point operations to calculate the geometrical centre of all solute charge groups and the number of floating point operations to calculate the distance between one charge group pair. The complexity of distance calculations also depends on the kind of periodic boundaries (*ThrG2*: rectangular; *Thr2*: octahedron).

GROMOS96 is delivered with a "brute force neighbour searching" algorithm (*nbnone*), where all possible (distinct) particle pairs are tested against the cutoff criterion. In the following this brute force pairlist algorithm is referred to as *oldPair*. A more sophisticated algorithm which is based on grid cells (section 4.1.3) is referred to as *newPair*. The brute force approach leads to the complexity model for *oldPair* as follows:

$$O(\text{oldPair}) = \frac{\text{nragt} \cdot (\text{nragt} + 1)}{2} \tag{3.6}$$

The memory demand is composed of the input coordinates of all atoms, the charge group positions as an intermediate result, and the pairlist itself as output. The pairlist is stored in two integer arrays: a pointer list 1) with nragt elements containing an index per charge group which is used to access its neighbours in the neighbour list 2). The size NN of the neighbour list depends on the cutoff radius and may be determined by experimentation or analytically by assuming a homogeneous liquid (3.10).

$$M(\text{oldPair}) = \text{ndim} \cdot (\text{nattot} + \text{nragt}) + \text{nragt} + \text{NN} \tag{3.7}$$

$$B_{\text{In}}(\text{oldPair}) = \text{ndim} \cdot \text{nragt} \tag{3.8}$$

$$B_{\text{Out}}(\text{oldPair}) = \text{nragt} + \text{NN} \tag{3.9}$$

$$\text{NN} = \frac{2\pi}{3} \cdot \frac{\text{nragt}^2 \cdot r_c^3}{V_{\text{box}}} \tag{3.10}$$

The accuracy of the NN-model is of fundamental importance, because the execution time of nonbonded forces calculation principally depends on the size of the pairlist. When simulating liquids, typical cutoff radii always lie between 0.6nm and 2.0nm. Thus the NN-model should be accurate for cutoffs within this range. The deviation of NN according to (3.10) is +3% for a 0.6nm cutoff and +0.1% for a 2.0nm cutoff, so the introduced error due to density fluctuations near the solute may be neglected. As subsequent function models are separated into solute and solvent parts, the pairlist model must necessarily follow this concept. The neighbour list has to be divided up into three lists, one list for solvent-solvent (subscript 'vv'), one for solute-solute (subscript 'ss'), and one for solute-solvent (subscript 'sv') interactions. Appropriate size models for these sub-lists are derived directly from (3.10) by substituting $\mathrm{nragt} = \mathrm{ncag} + \mathrm{nvcag}$ and $\mathrm{NN} = \mathrm{NN_{vv}} + \mathrm{NN_{ss}} + \mathrm{NN_{sv}}$:

$$\mathrm{NN_{vv}} = \frac{2\pi}{3} \cdot \frac{\mathrm{nvcag}^2 \cdot \mathrm{r_c}^3}{\mathrm{V_{box}}} \tag{3.11}$$

$$\mathrm{NN_{ss}} = \frac{2\pi}{3} \cdot \frac{\mathrm{r_c}^3}{\mathrm{V_{box}}} \cdot \mathrm{ncag}^2 \tag{3.12}$$

$$\mathrm{NN_{sv}} = \frac{2\pi}{3} \cdot \frac{\mathrm{r_c}^3}{\mathrm{V_{box}}} \cdot 2\mathrm{nvcag} \cdot \mathrm{ncag} \tag{3.13}$$

The deviation of $\mathrm{NN_{vv}}$ is approximately -5%, whereas the deviation of ($\mathrm{NN_{ss}}$+$\mathrm{NN_{sv}}$) is between +18% (2.0nm cutoff) and +50% (0.6nm cutoff). Deviation data is derived from the *ThrG2* benchmark and varies slightly with the solute to solvent ratio. The separated model delivers too many solute interactions at the expense of less solvent pairs. As solute interactions are more complex to calculate, these uncertainties may increase the estimated total execution time.

Instead of an analytical *newPair* complexity model, the execution times were determined by experimentation for cutoffs between 0.6nm and 2.0nm. The step size was chosen 0.1nm and the section between two adjacent cutoff radii has been interpolated. The algorithm and its performance compared with the brute force approach is described in section 4.1.3. The *newPair* algorithm delivers the pairlists in a different format: a third integer array with nragt elements was introduced containing the number of neighbours of each charge group. This additional information simplifies the access to neighbours of a given charge group.

$$\mathrm{M(newPair)} = \mathrm{ndim} \cdot (\mathrm{nattot} + \mathrm{nragt}) + 2 \cdot \mathrm{nragt} + \mathrm{NN} \tag{3.14}$$

$$\mathrm{B_{Out}(newPair)} = 2 \cdot \mathrm{nragt} + \mathrm{NN} \tag{3.15}$$

$$\mathrm{B_{In}(newPair)} = \mathrm{B_{In}(oldPair)} \tag{3.16}$$

Solvent-Solvent Nonbonded Interaction

Following the concept of the pairlist model, table 3.6 lists the number of floating point operations for one atom-atom interaction calculation. 'absr' and 'signr' are standard FORTRAN library functions. Type casts appear only in conjunction with the parallel version of the function, which is described in section 4.3.2.

Operation	octa, Virial	octa, no Virial	rect, Virial	rect, no Virial
+, −	(ndim)*7 + (ndim*p1)*2 + (ndim*p2)*1 + (p2)*1 + (l4d)*2 + 10	(ndim)*4 + (ndim*p1)*2 + (ndim*p2)*1 + (p2)*1 + (l4d)*2 + 10	(ndim)*7 + (ndim*p1)*1 + (l4d)*1 + 8	(ndim)*4 + (ndim*P1)*1 + (l4d)*1 + 8
*	(ndim)*4 + (p2)*1 + (p2*l4d)*1 + 14	(ndim)*2 + (p2)*1 + (p2*l4d)*1 + 14	(ndim)*4 + 14	(ndim)*2 + 14
/	1	1	1	1
√	1	1	1	1
absr	(ndim)*1	(ndim)*1	0	0
signr	(p2*ndim)*1	(p2*ndim)*1	0	0
type cast	(ndim)*2 + 6	(ndim)*1 + 6	(ndim)*2 + 6	(ndim)*1 + 6

Table 3.6 Number of floating point operations for one atom-atom interaction, function *solvSolv*. The columns apply to octahedron (octa) and rectangular (rect) periodic boundaries, with and without Virial calculation, respectively.

Atom-atom interactions are calculated for all distinct solvent charge group pairs according to the solvent pairlist. It is assumed that periodic boundary conditions always apply. The operations as they appear in table 3.6 are executed $\text{nram}^2 \cdot \text{NN}_{\text{vv}}$ times per MD time step, viz

$$\text{O}(\text{solvSolv}) = \text{nram}^2 \cdot \text{NN}_{\text{vv}} \tag{3.17}$$

The amount of input data depends on the control flow of the MD algorithm: only if a new pairlist was calculated within the current time step ('ldopair' == true), the respective amount of data appear on the function's input. In the opposite case, (3.18) reduces to the first summand: the coordinates of all solvent atoms. The output consists of the partial forces on the solvent atoms, the molecular Virial (if calculated), and the energies according to the simulation specification. For a reasonable problem size, Virial and energies do not contribute to B_{Out}, hence, they are omitted in the model.

$$\text{B}_{\text{In}}(\text{solvSolv}) = \text{ndim} \cdot \text{nvcag} \cdot \text{nram} + (2 \cdot \text{nvcag} + \text{NN}_{\text{vv}}) \cdot \text{ldopair} \tag{3.18}$$

$$\text{B}_{\text{Out}}(\text{solvSolv}) = \text{ndim} \cdot \text{nvcag} \cdot \text{nram} \tag{3.19}$$

$$\text{M}(\text{solvSolv}) = \text{B}_{\text{In}}(\text{solvSolv}) + \text{B}_{\text{Out}}(\text{solvSolv}) \tag{3.20}$$

Solute-Solute and Solute-Solvent Nonbonded Interaction

The complexity of this function (*soluForce*) is in proportion with the number of solute pairlist entries. A detailed floating point operation analysis similar to the solvent-solvent interaction model has not been done. Analogous to the solvent model represented by (3.17), equation (3.21) also assumes that atom-atom interactions within a solute charge group are not calculated in the nonbonded functions. In contrast to the solvent-solvent case, this is not necessarily correct for solutes, since calculating nonbonded atomic interactions within the same solute charge group is a GROMOS option. In most cases this option is disabled, and its contribution to the execution time would be very small, i.e. linear to the number of solute atoms.

$$O(\text{soluForce}) = \left(\frac{\text{nrp}}{\text{ncag}}\right)^2 \cdot \text{NN}_{\text{ss}} + \frac{\text{nrp} \cdot \text{nram}}{\text{ncag}} \cdot \text{NN}_{\text{sv}} \tag{3.21}$$

Note the subtle differences between the memory and I/O models of *solvSolv* and the solute nonbonded interactions *soluForce*: *soluForce* requires all atom coordinates as input, plus the pairlist depending on the control flow parameter 'ldopair'. The output consists of the partial nonbonded forces, other results do not contribute for systems of a reasonable size.

$$B_{\text{In}}(\text{soluForce}) = \text{ndim} \cdot \text{nattot} + (2 \cdot \text{nragt} + \text{NN}_{\text{ss}} + \text{NN}_{\text{sv}}) \cdot \text{ldopair} \tag{3.22}$$

$$B_{\text{Out}}(\text{soluForce}) = \text{ndim} \cdot \text{nattot} \tag{3.23}$$

$$M(\text{soluForce}) = B_{\text{In}}(\text{soluForce}) + B_{\text{Out}}(\text{soluForce}) \tag{3.24}$$

Virial and Remaining Functions

If the Virial is to be calculated, the centre of mass positions of solute charge groups are calculated in accordance with (1.17). For solvent molecules, the centre of mass is the position of the first atom of the molecule. For both solute and solvent, the relative positions of the atoms to the centre of mass positions must be calculated according to (1.18). The complexity of the Virial preparation function *prpvir* as well as its memory and I/O demand is in proportion to the number of atoms. Complexities as well as the memory and I/O demand of the remaining functions (bonded forces, integration, etc.) are linear with the number of atoms. Simulation setup, program initialization and termination, and file I/O (e.g. writing trajectories or other data) is not considered and not modelled.

3.3 Modelling GROMOS for Design Space Exploration

3.3.1 Parallel Model Extension

Profiling the Gromos application provides information about performance bottlenecks and the most computation time consuming functions. In the previous section, a behavioural model for all functions has been developed including the complexity of the function, the memory requirement and I/O demands. Most accurate models are provided for the most time consuming functions. Dependence graph models—introduced in section 2.2.6—represent control or data dependences of an algorithm and, in addition, task level concurrency. This type of parallelism may *only* be applied on the level of complex functions. Considering the GROMOS MD algorithm, the conse-

quence is that the most time consuming function *solvSolv* may be executed in parallel with other functions, e.g. *bonded*, but cannot be further decomposed. To overcome this drawback, adapting the functional model seems appropriate.

A first simple approach would duplicate the function within the dependence graph as many times as it seems appropriate. Then, each instance of *solvSolv* would have its own function parameters, this way allowing maximum flexibility by means of load balancing and decomposition method. In addition, communication requirements and functional complexity are associated with each instance independently. Once the number of instances for a specific function is fixed, and the functional parameters are calculated in accordance with the parallelisation strategy, the dependence graph is reluctantly changed. Hence, if the grade of parallelism is to be changed, a new specification graph is needed. This is in contradiction to synthesis-based design space exploration, where it is of central interest to test many different approaches semi-automatically and in a short time. A dependence graph, where functions are allowed to represent scalable *parallel function nodes*, overcomes this drawback.

In addition to the task level parallelism further parallelism is introduced into the system. This new kind of parallelism is an attribute to functional nodes in the specification. The execution semantics of the graph do not change, and the specification remains compatible with the synthesis tool. In return some restrictions arise: 1) because the synthesizer is not able to process this kind of parallelism, it is not specified explicitly in the dependence graph model; 2) the parallelisation strategy must take into account that a function can only be decomposed if it is mapped onto identical computational nodes. Thereby a symmetric decomposition is enforced and a linear scaling is used to modify the (sequential) function parameters (table 2.1). Due to the linear scaling approach the number of identical processing nodes, on which a *parallel function node* is executed, is independent of the functional model itself. The parallel attribute of a functional node is of type boolean, and the grade of parallelism is not a property of the behavioural function description. Instead, the number of identical processing instances is a property of computational nodes of an architecture and is included in the models after the binding of functions to hardware resources.

Assuming a parallel function node $v_j \in V_D$ mapped onto a computational node $v_i \in V_A$. The computational node itself—represented as a single node in the architecture graph G_A—consists of P_i identical processing elements. Then, the *parallel execution time* is given by (3.25).

$$T_j^{comp,p}(v_i) = \frac{Load_j}{P_i} \cdot \frac{O_j(\mathbf{n})}{Perf_j(v_i)} \tag{3.25}$$

In contrast to equation (2.1), the parallel execution time models the execution time of the 'slowest' virtual processing element. Although all virtual processors are identical and a linear scaling is assumed, the parameter '$Load_j$' allows to model workload fluctuation or parallelisation overheads. If the load balancing is perfect and overheads are neglected, '$Load_j$' is set to one. Later on, the parallel extension of the execution time is employed for system synthesis and for performance estimation.

The communication time from and to a parallel function node changes in a similar way, but depending on the parallelisation strategy. For a domain decomposition with replicated data, the communication time must be multiplied with P_i, if broadcast and collective communications are modelled as successive point-to-point transmission.

3.3.2 Scaling the GROMOS Models

All generic GROMOS function models developed in section 3.2.4 are completely hardware independent. They specify the number of elementary operations in dependence upon problem parameters. In addition, elementary operations of the functions *oldPair* and *solvSolv* are expressed in terms of the number of floating point calculations per elementary operation. Once a function is mapped onto a processing node (binding), the respective execution and communication times may be calculated (equations (2.1) and (2.2)). This is where the hardware dependence comes in.

Although the models are scaled with measured execution times of one typical MD simulation, it is possible to predict the performance of arbitrary simulations, assuming that the models cover all relevant simulation parameter dependences (table 3.4). Once the models are scaled with a typical MD simulation (e.g. *ThrG2*), the result must be verified. Verification is done by modification of typical MD simulation parameters, e.g. the cutoff. Cross-checking the scaled model with another molecular system is recommended. In the course of the following discussion it will be explained how the models are scaled under different circumstances, and an example is given for the function model *solvSolv*.

Given the complexity $O_j(\mathbf{n})$ of a function v_j; $O_j(\mathbf{n})$ providing the number of elementary operations. Assuming that the function v_j is mapped onto the processing node v_i, three scenarios to determine $T_j^{comp}(v_i)$ or $Perf_j(v_i)$ are differentiated:

1. v_i is available for profiling the function v_j.
   - determine the execution time of v_j on v_i (measurement) and the overhead o_i.
   - calculate $Perf_j(v_i)$ according to (2.1)
2. v_i is *not* available for profiling, but standard benchmarks are available.
   - choose a benchmark of v_i which is expected to coincide best with the function v_j. Estimate o_i.
   - run or obtain the same benchmark on a reference machine. It is an advantage for the reference machine to have a comparable architecture as v_i. Calculate the ratio of the benchmark results.
   - run v_j on the reference machine, scale the measured execution time with the benchmark ratio, this way estimating the execution time of v_j on v_i.
   - calculate $Perf_j(v_i)$ according to (2.1)
3. the number and kind of floating point operations per elementary operation is known, v_i is *not* available for profiling, and benchmarks of v_i are *not* available.
   - refer to the data sheet of v_i to determine the number of clock cycles for the floating point operations. Determine the number of cycles per elementary operation under consideration of possibly available multiple floating point units.
   - add overhead cycles: estimate the efficiency of an implementation. This is the gap between the delivered floating point performance compared with the peak performance.
   - multiply the number of elementary operation with the number of clock cycles.

- calculate $T_j^{comp}(v_i)$ by dividing the total number of clock cycles by the operating frequency of the processor. If required, consider an overhead o_i.

If some functions are executed in parallel on identical processing nodes, the sequential execution time must be corrected according to (3.25). Regardless of the scaling strategy it is generally assumed that there is always enough main memory so that hard disk swapping never occurs.

Scaling the problem dependent part of the memory demand is easily done after fixing the number of bytes per word. The execution time of communication nodes $v_j \in V_D$ is determined by equation (2.2). The communication parameters are derived from data sheets, or from the specification of the communication link. In special cases estimations are sufficient e.g. if two successive functions are mapped to the same computational node, and only internal communication occurs. The memory demand was not taken into account in order to explore the GROMOS96P design space (section 3.4).

Example: solvSolv

The scaling procedure is illustrated with *solvSolv*. The function model is scaled with the *ThrG2* molecular system (table 3.6, with Virial) on a SunUltra30 workstation (appendix B). Table 3.7 summarizes the required simulation parameters to calculate the size of the pairlist, which is part of the *solvSolv* complexity model.

Parameter	Value	Derived Quantities
nram	3	equation (3.11): $NN_v = \frac{2\pi}{3} \cdot \frac{nvcag^2 \cdot r_c^3}{V_{box}} = 1{,}859{,}739$
nvcag	10,961	
r_c	1.4	table 3.2: $T^{comp}_{solvSolv}(SunUltra30) = 19.52s$
V_{box}	371.271	

Table 3.7 Scaling example for the SunUltra30, function *solvSolv*: problem parameters and derived quantities. Values for simulation parameters are derived from table A.1, the execution time from table 3.2.

With the measured execution time, the performance model for *solvSolv* on the SunUltra30 is determined (3.26). In this context, [EOP/s] is the number of atomic solvent-solvent interaction calculations per second.

$$Perf_{solvSolv}(SunUltra30) = \frac{nram^2 \cdot NN_{vv}}{T^{comp}_{solvSolv}(SunUltra30)} = 857{,}462\frac{EOP}{s} \tag{3.26}$$

Table 3.8 provides the missing parameters which are required for an additional floating point analysis to asses the quality of the implementation. The first two columns of table 3.8 list the number of CPU clock cycles per floating point operation for single (sp) and double precision (dp) floating point arithmetic. Latencies of addition (add) and multiplication (mult.) are three cycles each, whereas division (div.) and square root operations (sqrt) are not pipelined.

Cycles	sp	dp	Cycles	sp	dp	Parameter					
add	1	1	div.	12	22	ndim	3	l4d	0 (false)	p1	0.13[a]
mult.	1	1	sqrt	12	22	p1	0.13	ldovir	1 (true)	p2	0.14[b]

Table 3.8 Scaling example for the SunUltra30, function *solvSolv*: the number of CPU clock cycles per operation for single (sp) and double precision (dp) floating point arithmetic. Additional parameters are given as they are required for floating point analysis in accordance with table 3.6.

a. Measured with *ThrG2* benchmark, refer to [42]

b. Measured with *Thr2* benchmark, refer to [42]

The values for the number of cycles are derived from the UltraSparc-I microprocessor data sheet[1] and are used to calculate the number of floating point operations per atom-atom interaction and the number of clock cycles per atom-atom interaction (table 3.9).

Parameter	Value	Unit	Derived from
number of floating point operations per atom-atom interaction	58	[FLOP]	table 3.6 (56 add/multiply + 2 divide/square root)
number of clock cycles per atom-atom interaction	100	[cycles]	table 3.8 (56 add/multiply + 44 divide/square root)
time per atom-atom interaction	338	[ns]	$\frac{100\text{cycles}}{296\text{MHz}}$, table 3.9

Table 3.9 Scaling example for the SunUltra30, function *solvSolv*: calculation of the execution time of one atom-atom interaction using the floating point model.

The result is the best-case time for one atom-atom interaction calculation (338ns). Without adding any overhead cycles, the estimated execution time $T^{comp,est}_{solvSolv}$ is calculated by multiplying the time per interaction with the total number of interactions. The resulting estimated time (equation (3.27): 5.66 seconds) is about 3.5 times faster than the measured value (table 3.2: 19.52 seconds). As a consequence, predicting the execution time on the basis of floating point models is not recommended. If possible, runtime measurements or estimations based on a SPEC comparison are preferred. It could be shown in [45] that the uncertainty of performance estimations using SPEC ratios is in the range of ±10%.

$$T^{comp,est}_{solvSolv}(\text{SunUltra30}) = \text{nram}^2 \cdot \text{NN}_{vv} \cdot 338\text{ns} = 5.66\text{s} \tag{3.27}$$

All generic GROMOS96 models were scaled using runtime measurement results and the *ThrG2* benchmark with a 1.4nm cutoff. For 100 steps, the scaled model predicts an execution time of 2951 seconds, whereas the measurement delivered 2976 seconds (table 3.2). The deviation of less than one percent is due to the imbalance of the pairlist model for solute and solvent. More interesting is the comparison for a different cutoff radius (0.8nm): the model predicts an execution time of 1066 seconds, the measurement delivers 1074 seconds, thus leading to a deviation of also less than one percent.

1. Available at http://www.sun.com/microelectronics/whitepapers/UltraSPARCtechnology/

3.3.3 Models of the MD Algorithm

The dependence graph of the iterative time stepping MD algorithm is shown in fig. 3.1a, representing the algorithm which has been described in section 1.4.2. The specified algorithm does not correspond to the initial sequential GROMOS96 version, but rather represents the software structure of the new parallel implementation. The graphic editor of the Codesign framework [10] was used because of its interface to the system synthesis tool [7].

Parallel function nodes as introduced in section 3.3.1 are not explicitly shown on the dependence graphs of fig. 3.1. The reason is that the system synthesizer treats functional nodes in accordance with the execution semantics, which is the same for parallel functions nodes as for all nodes.

In fig. 3.1, complex functions are represented by shadowed, rectangular functional nodes, whereas communication is depicted as white, round communication nodes. The functions *soluPair*, *solvPair*, *soluForce* and *solvSolv* are parallel function nodes. For the sake of simplicity, some algorithmic functions were put together. This is appropriate if it is known in advance that the mapping of a group of functions is the same anyway. If the Virial is to be calculated, the solute part of the Virial preparation—which is not parallelisable—is included in the function *integration*, whereas the solvent part—which is parallelisable—goes to *solvSolv*. The function *parts* sorts the charge group spatially in preparation for the pairlist construction. In contrast to GROMOS96, the pairlist construction is decomposed into two functions *soluPair* and *solvPair*, both of which delivering the pairlist for *solvSolv* and *soluForce* respectively. Due to this decomposition, and because *oldPair* is not parallelisable, the algorithm *newPair* (section 4.1.3) must be used. Model parameters for *soluPair* and *solvPair* are easily derived from the *newPair* model. The function *sum* collects and adds partial forces, thus its functional model is trivial. Finally *integration* incorporates not only the Leapfrog integrator, but also the *shake* algorithm and all remaining functions within the main MD loop.

The dependence graph of fig. 3.1b represents exactly the same algorithm with the difference that some redundant functional (light grey, *F3*, *F4*, *F9*, *F10*) and communication nodes (*c11*, *c12*, *c20*, *c21*) have been inserted. Redundant functions nodes always have the complexity zero, thus the execution time is always zero, independent of the mapping. The function *join* determines the sum of the partial forces delivered by *solvSolv* and *soluForce*. Additional nodes are inserted to enable the system synthesizer exploiting a maximum number of feasible bindings—if the mapping edges in the specification graph provide enough flexibility.

The amount of data that a certain communication node is carrying is determined by the I/O specification of the connected functional model. For example, the input of the function *solvSolv* is composed of the pairlist delivered by *solvPair* and the particle coordinates from *integration*. The nodes *c1*, *c2*, *c3*, and *c4* (referring to fig. 3.1a) all represent the communication of particle coordinates. If all target functions of these communications are mapped onto the same computational node, the model considers four times the same data transmission. This is a weakness of the specification model, because repetitively communicating the same data to the same target node would never be done in practice.

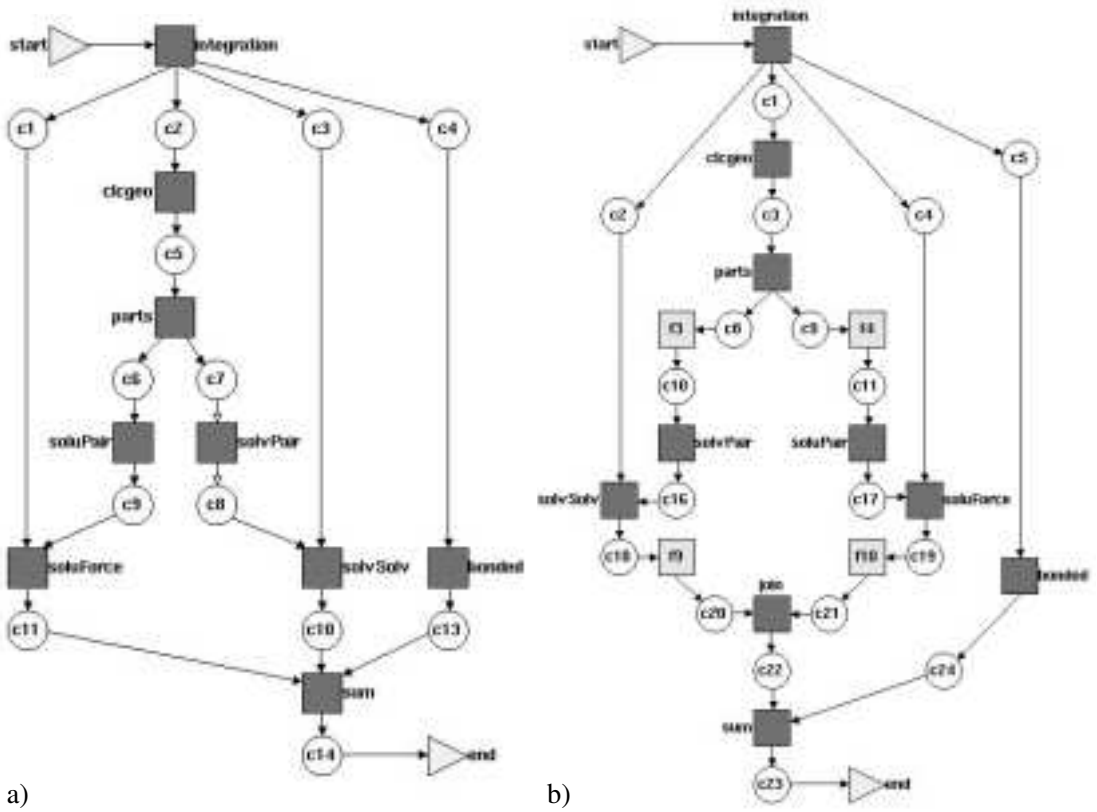


Figure 3.1 Dependence graph of the parallel GROMOS MD algorithm:
a) Basic algorithm.
b) Extended graph with redundant nodes for system synthesis.

3.3.4 Architecture Models

Two types of architecture models are separated: the abstract graph model which is used for synthesis, and block diagrams. The block diagrams in fig. 3.2 are not used for automated processing but describe concrete implementations for which the performance is estimated. All architectures have in common that they are connected to a host workstation (SunUltra30) with the PCI (Peripheral Component Interconnect) bus. The dashed line indicates the board outline of dedicated hardware approaches. Modern workstations can be extended with up to six PCI cards, that way enabling all depicted systems to be scalable. The PCI bus is modelled with a sustained bandwidth of 30 MBytes per second (MB/s), at a latency of 70µs per message, and with a message size of 2kByte (kB). The on-board communication (bus) is fixed to 50MB/s with a small latency (3µs per 128 Byte message). The bus model applies for all architectures with a shared bus.

The first architecture (fig. 3.2b) consists of six ASICs especially designed for the function *solvSolv*. The study of existing ASIC-based MD accelerators (section 3.1) has shown that six force processors, all accessing the same bus, is a reasonable number. The host interface (HI) of this architecture is also a specialised processor generating a pairlist (PL) if required. The pairlist is used on the host interface to distribute the particles on the board, since each ASIC has a private

particle and partial forces memory. Considering some glue logic devices, the physical extent of a PCI board would not allow more than seven computational nodes. The performance of one ASIC was assumed to be two times the performance of the host. This is in conjunction with third party developments, e.g. the MD Engine [63] (section 3.1.1). Referring to (3.26) and to the MD Engine performance (400ns per nonbonded interaction calculation), it was assumed that a GROMOS dedicated chip reaches at least the same performance as the MD Engine chip. Because the pairlist processor has to handle additional tasks like data distribution, its execution time is assumed to be equal to that of the host.

A similar approach incorporates six general purpose RISC (reduced instruction set computer) microprocessors on one board (fig. 3.2b). Analogous to the ASIC-based approach, memory bus congestion is reduced since each processor has a local memory. The difference to the previous approach is that all parallel MD functions can be executed on the coprocessors. Therefore, a dedicated pairlist processor is not needed. Fig. 3.2b represents two architecture alternatives, differing in the communication structure: the first proposal allows the processors to communicate with direct neighbours over a bidirectional ring. This is advantageous if the MD algorithm is parallelised following the domain decomposition scheme (section 4.2). The alternative architecture has a shared bus for interprocessor communication. The host interface (HI) either distributes the data over the ring or via the bus.

Fig. 3.2c illustrates a solution without dedicated hardware: a number of general purpose processing elements (PE) are interconnected with a high-speed network, e.g. Myrinet [33]. All parallel MD functions may be mapped onto the PEs, because general purpose clients like PCs or workstations are used. The performance of the clients and the host machine shall be the same. The Myrinet bandwidth is assumed to be very similar to the PCI bus: 30MB/s, full-duplex, with a $50\mu s$ latency per 2kB message. The switch introduces an additional latency of $20\mu s$ per message.

Myrinet performance information for message passing systems may be derived directly from Myricom Inc.[1], or from published experimental results [33][45][56]. Bandwidth and latency of a Myrinet network are strongly dependent on the message size, and the maximum bandwidth (100MB/s) is achieved with message lengths of about 100kB and a latency around $100\mu s$.

The hierarchical DSP architecture of fig. 3.2d consists of thirteen 100 MHz Sharc DSP processors [2] arranged on one board. A high level cross compiler as well as a Sharc simulator is available from the manufacturer [3]. Therefore, all functions may be executed on all processors. Each DSP processor has enough on-chip memory (4 MBit), so external memory is not needed if domain decomposition is applied. An integrated I/O processor with DMA provides six 8-bit link ports for interprocessor communication with a maximum throughput of 600MB/s. Consequently, the proposed architecture uses three upper level Sharc processors to distribute and collect the data (fig. 3.2d). Remaining ten DSPs are planned for pairlist and forces calculation. Due to the fact that all processors are interconnected with link ports, the amount of additional glue logic is reduced to the host interface (HI), and 13 Sharc chips per board is a reasonable number. The link port communication model assumes a sustained bandwidth of 80MB/s, with a small latency of $100ns$ per 64 Byte message.

1. Myricom, Inc., 325 N. Santa Anita Ave., Arcadia CA 91006: performance results for a wide range of machines and network topologies as well as various third-party device drivers are available: http://www.myri.com

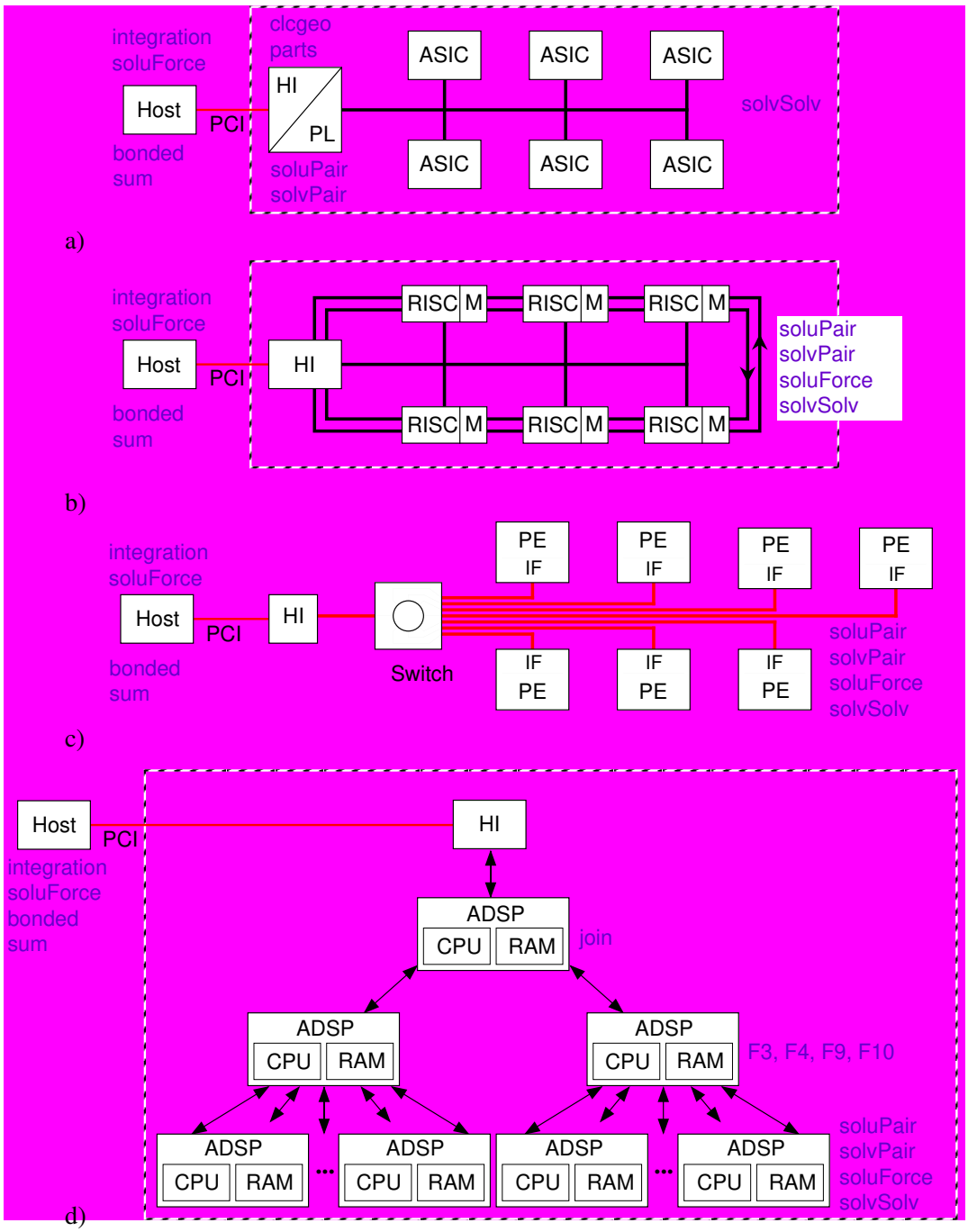


Figure 3.2 Architectural block diagram models: four architectures which are referred to as ASIC, RISC, Net and Sharc:
a) Distributed memory coprocessor board consisting of six ASICs on a shared bus.
b) Distributed memory coprocessor board with off-the-shelf RISC microprocessors. Processors communicate either over a shared bus or a bidirectional ring.
c) Net: Workstation cluster with a switched high performance network (e.g. Myrinet).
d) Hierarchical Sharc-DSP [2] coprocessor.

Investigations with a predecessor 40 MHz Sharc version have shown that about 25% of the peak performance (Perf = 40 MFLOPS) can be reached: In [45], the *ThrG2* pairlist construction with the brute force algorithm has been implemented on the 40 MHz Sharc for testing purposes. The measured execution time was 86.2 seconds. The execution time of the model (table 3.5) is calculated with equation (3.6) using the peak performance value (11 floating point operations without adding extra cycles: 20.6 seconds). In order to estimate the performance of the new Sharc processor, the duplicated floating point unit (factor 1.5) and the higher operating frequency of 100 MHz (factor 2.5) must be considered. With respect to the 40 MHz chip, an overall performance gain of a factor 3.75 was assumed. Under this assumption, the pairlist is calculated within 23 seconds on the new Sharc, whereas the host needs 30.35 seconds (table 3.2). This rough estimation was used to scale the performance of the Sharc processors to be 1.32 times faster than the host machine.

For system synthesis, the architecture graph must be a super-set of all architectures which can be allocated, and the knowledge of the designer is essential to provide suitable architectures. Two different super-architectures are defined as abstract architecture graphs G_A (fig. 3.3).

The first abstract architecture (fig. 3.3a) consists of a host workstation and three alternative coprocessor or client architectures: 1) an architecture consisting of specialised ASICs for force calculation 2) a network architecture with general purpose computational nodes PE_gp and a high performance network, and 3) a dedicated hardware approach with off-the-shelf microprocessors PE_ded. Fig. 3.3a represents a super-architecture consisting of the three concrete proposals of fig. 3.2a/b/c. All three proposals have in common that they are scalable ($P_i \geq 1$) by replicating the computational nodes named *Host*, *Asic*, *PE_gp*, and *PE_ded*. *Scalable computational nodes* are depicted rectangular, whereas those with rounded corners are not scalable ($P_i = 1$). The host workstation is connected to the client architectures with an expansion bus BUS1, e.g. PCI.The full-duplex Myrinet is modelled accordingly.

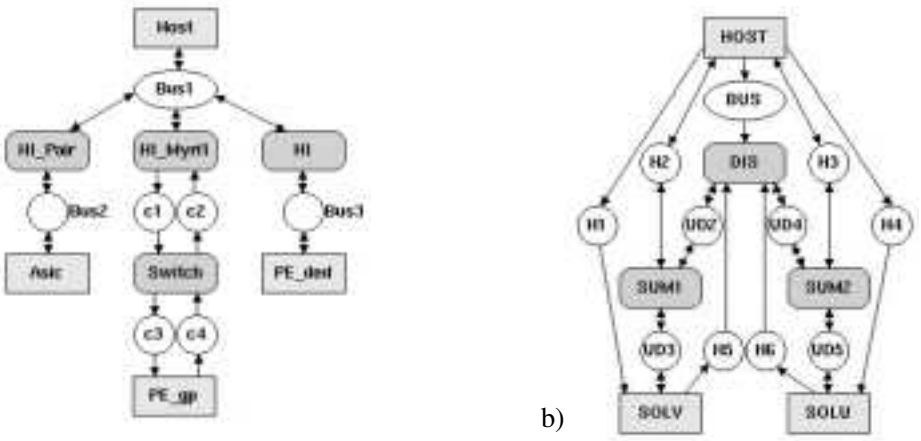


Figure 3.3 Abstract architecture graphs for system synthesis:
a) Three implementation alternatives in accordance with fig. 3.2a/b/c.
b) Hierarchical Sharc DSP-based architecture in accordance with fig. 3.2d.

Fig. 3.3b is the abstract architecture of the hierarchical Sharc board (fig. 3.2d). The scalable computational nodes SOLU and SOLV represent the lowest level of the DSP hierarchy. The nodes SUM1, SUM2 and DIS model intermediate levels, responsible for data distribution and calculating the sum of partial forces. Similar to the algorithm specification, and for the same reason, redundant communication nodes were inserted (H1, H2, H3, H4, H5, H6).

3.4 Exploring the Design Space

Some tasks to explore the design space for new GROMOS96P architectures have already been accomplished: the requirements list and the main goals have been stated in section 1.5.2. Existing parallel software approaches and hardware accelerators were also briefly assessed in the introductory part, and a more thoroughly analysis about pros and cons is outlined in section 3.1. Based on this knowledge, new architectures have been introduced in section 3.3.4. One design decision may be derived implicitly from the table of the best case speed-ups (table 3.3): to reach a speed-up of at least ten, it is necessary to parallelise the nonbonded forces calculation (solute *and* solvent interactions) *and* the pairlist construction. Coming from this situation, system synthesis is used to obtain even more design alternatives.

3.4.1 System Synthesis

The synthesizer is able to exploit different allocations and function mappings and to provide a appropriate schedule. Through varying cost and communication parameters of the previously proposed architectures, the number of degrees of freedom in the input specification is increased. Since the input for the synthesizer is a dependence graph, the control flow including pairlist calculation has been selected to determine the dependence graph (fig. 3.1).

Three examples are explored with the synthesizer: The dependence graph of fig. 3.1b is combined with the abstract architecture of fig. 3.3b to build the specification graph of the first two examples. The third example deals with the combination of fig. 3.1a and fig. 3.3a. All examples use the *ThrG2* benchmark to determine the execution times of algorithmic functions.

Example 1: Specification Graph composed of fig. 3.1b and fig. 3.3b

Associated with edges of the specification graph, which is not depicted due to its complexity, are execution times of communications and computations in accordance with the scaled function models of section 3.2.4. In principle, functional nodes of the dependence graph may be arbitrarily mapped onto computational nodes of the abstract architecture. Nevertheless, the designer will insert mapping edges carefully selected. Care must be taken when corresponding execution times are calculated: if a regular functional node v_j (regular: non-parallel or redundant) is mapped onto a scalable computation node v_i, equation (2.1) without parallel extension must be used in order to calculate the execution time, this way ignoring the parameter P_i, which is associated with v_i. If v_j is a parallel function node, the parallelism of v_i is taken into account by applying (3.25). On the other hand, if v_i is a regular node, which is the case if $P_i = 1$, equation (2.1) applies for all types of v_j.

The calculation of the communication time is more sophisticated: considering a parallel function node v_{j1} which is mapped onto a scalable computation node v_i consisting of P_i processors, and a communication node v_{j2} which feeds data to the binding (v_{j1}, v_i). In this case, the calcula-

tion of the communication time of v_{j2} depends not only on the mapping, but also on the decomposition scheme of the parallel function v_{j1}. For a domain decomposition (section 4.2), each processor of v_i works on disjunctive data sets. In this case it may be assumed that the communication time is constant for all P_i. With a replicated data decomposition approach v_{j2}, has to distribute all data to all instances of v_i. This is accomplished either with successive point-to-point communications or with a broadcast. The point-to-point strategy multiplies the communication time with the number of processors P_i, whereas an efficient broadcast may introduce only a certain overhead factor. The broadcast overhead factor is multiplied with the communication time instead of P_i. In the ideal case, when the broadcast is as fast as one point-to-point transmission, the broadcast overhead factor equals one. Because the calculation of all execution times is done manually, considering all cases is a lavish task and susceptible to errors.

Coming back to the example, the parameter P_i is fixed to five for *SOLV* and *SOLU*, respectively, and P_{HOST} is assumed to be one. The resulting abstract architecture represents exactly the block diagram of fig. 3.2d. The execution times of all functions are determined according to section 3.2 for a SunUltra30 host machine (*HOST*). Since the Sharc processors are estimated to be faster than the host machine by a factor of 1.32 (section 3.3.4), the execution times associated with corresponding mapping edges must be corrected. Further, if a parallel function node is mapped onto *SOLV* or *SOLU*, an additional speed-up of five is applied. The total execution time of *newPair* (fig. 4.7, 6.4 seconds) has been taken to approximate the time for *soluPair* and *solvPair:* according to the ratio of the respective pairlist sizes, *soluPair* takes one second on one Sharc processor, while *solvPair* takes five seconds, refer to equations (3.11), (3.12), and (3.13).

The functions *integration*, *bonded*, and *sum* may only be executed on the host. Additional mapping constraints are applied to parallel function nodes: *soluPair* and *soluForce* may not be executed on *SUM1* and *SOLV*, *solvPair* and *solvSolv* may not be executed on *SUM2* and *SOLU*. To ensure that the synthesizer finds at least one implementation, all nodes of the dependence graph are allowed to execute on host. Communication times are not scaled for parallel nodes, thus assuming either perfect broadcast communication or domain decomposition. The synthesizer is not able to handle floating point numbers. Therefore, execution times are normalised: all calculated times are multiplied with a factor of five and rounded to integers. Execution times of tasks running strictly on the host machine are intentionally enlarged for purposes of visualization.

The hardware cost for computational and communication nodes of the architecture graph is fixed as follows: the host costs 30, and each Sharc processor 4. These cost values are also normalised, thus are given without unit. Communication resources are free with the exception of redundant *H*x-nodes, which cost 1. Running the synthesizer with this specification results in a set of implementations in the cost–performance design space. Table 3.10 summarizes cost, performance and the allocation of the selected Pareto points. The time is given in units of seconds, on the expanded time scale as mentioned before. The synthesizer generates a schedule for each Pareto point. The schedule of fastest implementation (indicated bold in table 3.10) is shown in fig. 3.4.

Referring to table 3.10, the architecture with the overall cost of 62 has the same allocation of computational nodes as the fastest implementation with cost 64. The difference in the expense of both solutions is caused by the allocation of two additional *H*x nodes. The wide discrepancy between the performance is symptomatic of the behaviour of the synthesizer: the Evolutionary Algorithm (EA) is not able to find the communication path via an intermediate computational node, although for that reason redundant nodes were inserted in the dependence graph. Instead,

the EA determines the mapping in a way that time consuming tasks are executed on slow nodes. Even if the hardware cost and execution time of regular nodes is artificially reduced, the EA prefers the more expensive and slower redundant communication hardware. This results in the appreciably worse performance.

Specification	Cost	Time	Allocation
architecture: fig. 3.3b algorithm: fig. 3.1b	30	156	HOST only
	42	237	HOST, DIS, SUM1, SUM2
	62	219	HOST, DIS, SUM1, SUM2, SOLV
	64	**38**	HOST, DIS, SUM1, SUM2, SOLV

Table 3.10 Synthesis results of example 1, given as Pareto points. The fastest implementation is indicated bold. Note that not all computational resources are allocated to reach the maximum speed.

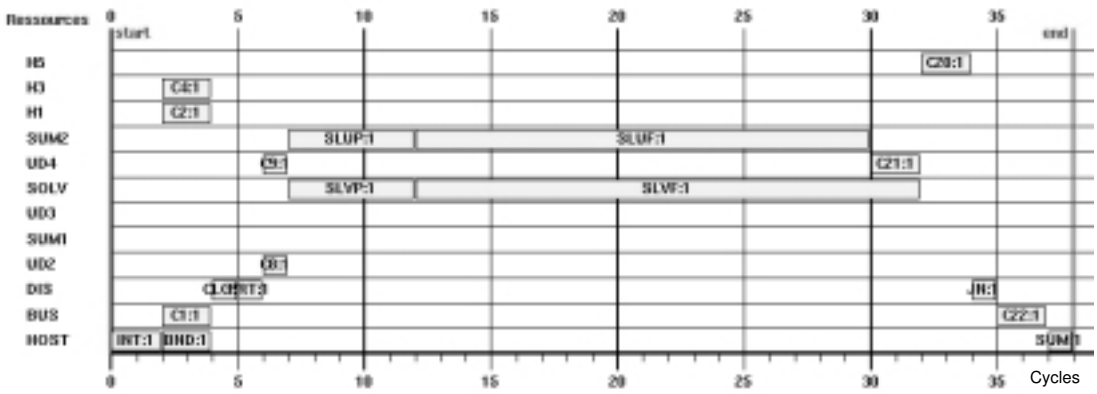


Figure 3.4 Schedule of example 1, representing the fastest implementation of table 3.10. The hierarchical Sharc architecture without the scalable computational node *SOLU* was allocated. To improve the readability, some function names are shortened: SLUP = *soluPair*, SLUF = *soluForce*, SLVP = *solvPair*, SLVF = *solvSolv*, INT = *integration*, BND = *bonded*.

Another motivation to introduce redundant nodes is to enlarge the population of the EA, and in this way the number of possible feasible bindings. Due to redundant nodes, the EA is able to find more and in particular better implementations, because a disjunctive population of individuals increases the probability that the EA finds new feasible bindings. The initial idea of steering the algorithm such that the EA itself chooses an implementation without redundant nodes did not succeed.

The surprising result of this first run is that *SOLU* is not allocated. The reason is that the ratio of the execution times of *solvSolv* and *soluForce* (table 3.2) is approximately 5:1. Not surprising, the function *soluForce* requires the same amount of time on the one Sharc *SUM2* as *solvSolv* on five *SOLV* Sharcs (fig. 3.4). As a result, the Evolutionary Algorithm has found the optimal architecture, binding, and schedule. This success must be considered with the reservation that the pro-

posed implementation is only optimal for the *ThrG2* benchmark, with a cutoff of 1.4nm, and if the pairlist is calculated. Since the capabilities of the EA do not go so far, the question remains whether or not this implementation is also well suited for arbitrary MD simulations.

Example 2: Specification Graph composed of fig. 3.1b and a reduced Architecture

The architecture of fig. 3.3b is modified by removing one branch of the hierarchy. The remaining nodes *SUM1* and *SOLV* represent two and ten Sharcs, respectively. This is simply accomplished by adapting the parameter $P_{SUM1} = 2$ and $P_{SOLV} = 10$, and the cost accordingly. This architecture also represents the Sharc hierarchy as depicted in fig. 3.2d with 13 processing nodes. The difference to the previous example is that the mapping of functions is less constrained. The parallel function nodes *solvPair*, *solvSolv*, *soluPair* and *soluForce* may be executed on SUM1 or SOLV. The execution times of the parallel functions are adapted accordingly, all other parameters are the same as for example 1. Running the synthesizer delivers implementations which are listed in table 3.11.

Specification	Cost	Time	Allocation
Architecture: fig. 3.3b without *SUM2* and *SOLU*. Algorithm: fig. 3.1b	30	156	HOST only
	42	87	HOST, DIS, SUM1
	82	75	HOST, DIS, SUM1, SOLV
	84	**26**	HOST, DIS, SUM1, SOLV

Table 3.11 Synthesis results of example 2, given as Pareto points. The fastest implementation is indicated bold.

The results show that for the fastest implementation all computational resources are allocated. The overall workload of the coprocessors is better than in the previous example (fig. 3.4 and fig. 3.5), where one Sharc (*SUM1*) was allocated exclusively for data distribution and partial sum calculation. Here, a better load balancing and more processing power leads to a faster but also more expensive solution. Again, it is not ensured that the proposed architecture performs well for other MD parameter sets.

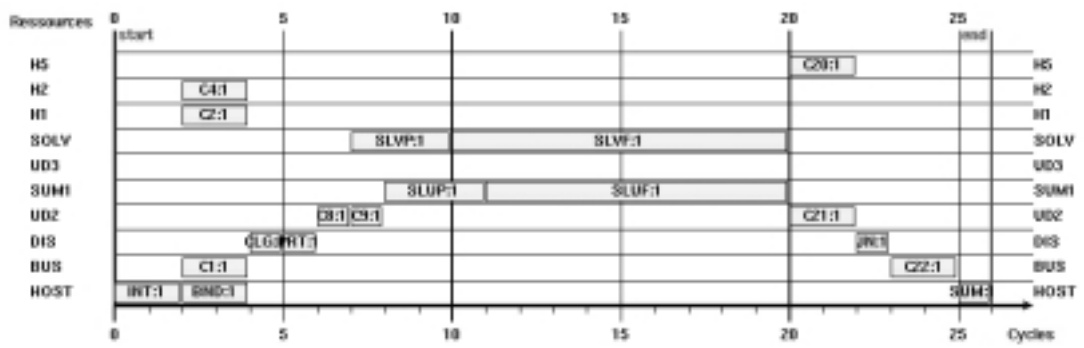


Figure 3.5 Example schedule of the fastest implementation of table 3.11. The allocation represents a hierarchical Sharc architecture consisting of 13 coprocessor Sharcs.

Another implementation not appearing in table 3.11 is rather evident, namely calculating all parallel functions on *SOLV*. In this case, the normalised total execution time would increase to 29. The functions *soluPair* (SLUP: plus one on *SOLV*) and *soluForce* (SLUF: plus two on *SOLV*) would contribute extra time because they must execute in sequence with SLVP and SLVF. The cost for this implementation would remain the same, since the allocation did not change. As this solution is not a Pareto point it is omitted in table 3.11.

Example 3: Specification Graph composed of fig. 3.1a and fig. 3.3a

No redundant nodes are inserted into the specification graph consisting of the dependence graph of fig. 3.1a and the abstract architecture of fig. 3.3a. Synthesis with the EA results in useless new implementations. The EA even does not find the fastest solution by allocating all three possible branches of the architecture, since functional nodes may not be mapped onto *PE_gp*. At least, depending on the grade of parallelism of *Asic* and *PE_ded*, the synthesizer correctly distributes the workload: solvent-related functions are mapped to *Asic*, the construction of the solute pairlist and the solute interaction calculations are mapped to *PE_ded*.

Changing the grade of parallelism means adding or removing dedicated hardware boards from the system, or adapting the parallel system to a certain simulation. This is a contradiction to the design goal stated in section 1.5.2: a *flexible* solution is to be found which reaches the required speed-up for a defined *range* of MD simulations (table 3.3).

If the system synthesizer is used to generate implementations, there are basically two possibilities to verify that a design proposal succeeds: 1) many benchmarks covering the required range of simulations must be explored with system synthesis 2) the behaviour of a the design proposal is estimated for all required simulation parameter sets. To find the GROMOS96P target architecture, the performance of the most promising suggestions is estimated by an implementation of analytical models.

3.4.2 Performance Estimations

The parallel computation model has been implemented in Mathematica [24] for performance estimation. As opposed to the synthesis approach, the main advantage of the Mathematica implementation is that control flow is considered and that generic functional models may be used. The current Mathematica estimation model stands out for the following features:

- interprocessor communication is modelled with the modified logP model (section 2.2.4), including bandwidth, latency and message length. The latency is not considered in dependence upon the message length, and the logP communication overhead (parameter o) is neglected.
- Scalable computational nodes are modelled in accordance with (3.25). All examined architectures have only one scalable computational node with $P_j = P_{tot}$.
- The parameter $Load_i$ is assumed to be one for all parallel function nodes (as perfect load balancing is expected). Parallelisation overhead is neglected for parallel function nodes.
- Algorithmic functions are implemented using generic models for the complexity and I/O demands, the memory demand is not modelled (section 2.2.7).
- Functional nodes are statically mapped onto computational nodes and scaled according to (3.25), and in accordance with section 3.3.2 (scaling)

Generic GROMOS models are scaled on the host machine, a SunUltra30. Four architectures are modelled and referred to as ASIC (fig. 3.2a), RISC (fig. 3.2b), Net (fig. 3.2c) and Sharc (fig. 3.2d). These architectures are described in section 3.3.4, including communication parameters and performance models of computational nodes. The default settings for the PCI bus, the Myrinet and on-board buses are according to section 3.3.4. The information about how the MD algorithm is implemented on these architectures for performance estimation is given here.

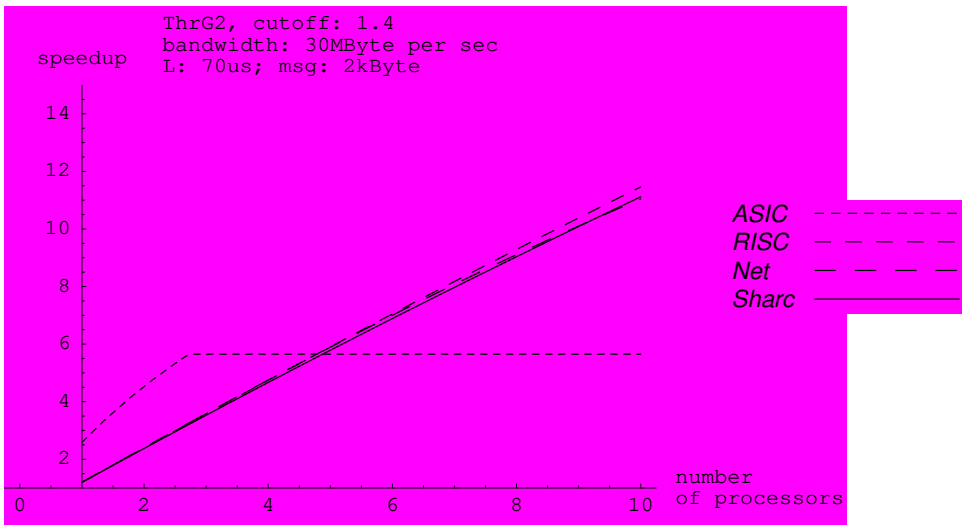
For each architecture a static assignment of functional nodes to computational nodes is determined. The mapping is also given in fig. 3.2, where the respective function names appear close to the processing nodes. Because the binding of functions to resources is fixed, a schedule may be determined considering the task level concurrency of fig. 3.1a, with one control flow extension. The control variable 'ldopair' steers the pairlist construction, and the pairlist update ratio is also a parameter in the Mathematica model. As a consequence, two schedules per architecture are required, one incorporating the pairlist and one for the interaction calculation only.

The implemented schedule for the ASIC architecture calculates the solvent pairlist on the coprocessor board, whereas the solute pairlist is calculated on the host in parallel. Pairlist construction and interaction calculation may not overlap, because the pairlist determines which particle pairs currently interact. When the new particle positions are determined after the integration step, the coordinates are downloaded to the ASIC architecture. The time required for the download of the solvent atomic coordinates is *not* dependent on the number of ASIC boards in the system, because each board reads in only the required data set. The data sets are defined by the total number of interaction processors (*number of processors*) P_{tot} in the system. The simulation box is divided up into P_{tot} slices of equal size. During the download process, each processor receives and stores only coordinates of atoms residing in the associated slice. During calculation, missing data is communicated from neighbouring processors in compliance with the domain decomposition (section 4.2). Since communication between boards is avoided, additional memory must be planned on each board containing atomic coordinates and partial forces of the spatial overlap zone. The bus is modelled appropriately.

The communication of interaction forces back to the host machine may not be accomplished by a broadcast, since data sets of different boards are not disjunctive. Thus, the communication from ASIC boards to the host is modelled as point-to-point communications. This leads to the sharp points in the speed-up estimation of fig. 3.6. The effect is only visible for the RISC and Sharc architecture, because the maximum performance of the ASIC solution is already reached with one board.

The main difference between the RISC and ASIC architectures is the function mapping: the RISC architecture with general purpose processing nodes allows all parallel function nodes to be executed on the coprocessor boards. This has a direct impact on the schedules. The data distribution algorithm as well as the decomposition scheme of the MD algorithm is not changed with respect to the ASIC architecture. If a pairlist is to be calculated, at first the local neighbour table for solute and solvent is constructed on each node. The functions clcgeo and parts are executed on each processor, although these functions are not parallel function nodes. The sequential execution times of *clcgeo* and *parts* apply. After all interactions have been calculated, the forces are read from the host, as is the case for the ASIC architecture. As mentioned in section 3.3.4, the communication between two adjacent processors is alternatively accomplished by a bidirectional ring. Both methods—bidirectional ring and bus—have been implemented in the Mathematica model.

a)



b)

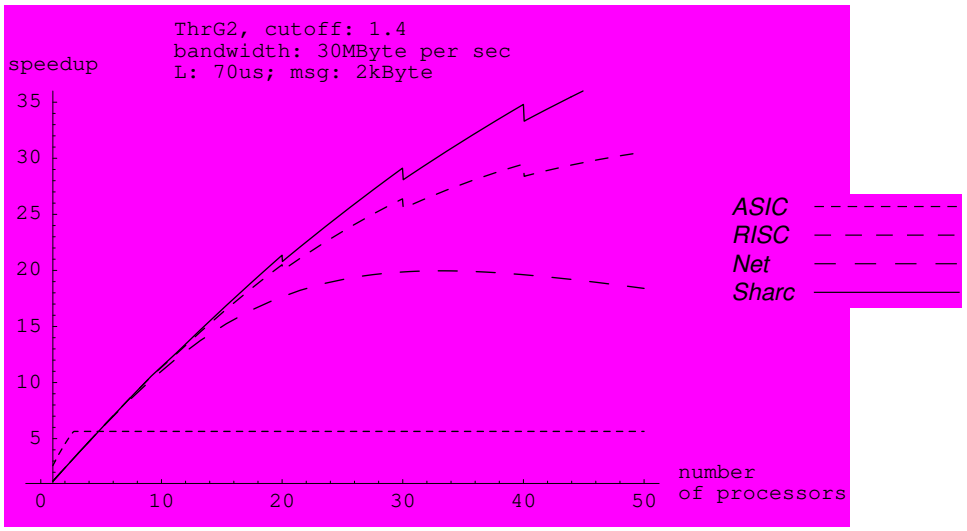
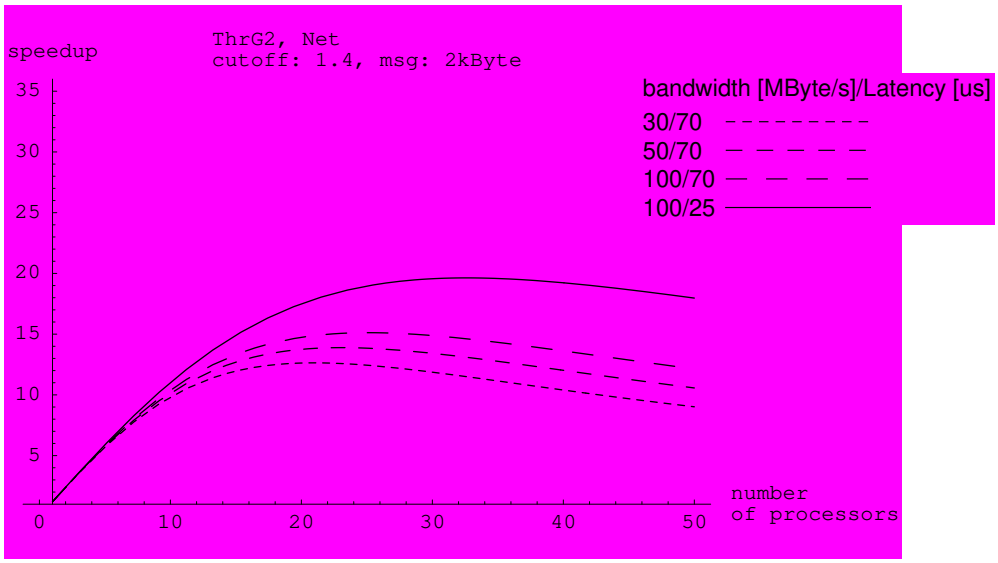


Figure 3.6 Speed-up estimation of parallel architectures of fig. 3.2, achieved with respect to the sequential GROMOS96 (table 3.2). Pairlist update every five steps. Indicated are values for bandwidth, latency (L) and message size (msg), applying to the PCI bus and the *Net* network. ASIC, RISC and Sharc architectures employ domain decomposition. *Net* uses the replicated data method with broadcast and collective communication. IBM SP-2 parameters are used for broadcast/collective communication.
a) Speed-up for 1-10 computational nodes.
b) Speed-up for 1-50 computational nodes.

The ring bandwidth was assumed to be 20MB/s with a latency of 20µs per 2kB message. Estimations have shown that the gain of the ring is insignificant if the ring contains less than about 20 processors. For that reason, the alternative of ring communications was not further examined.

a)



b)

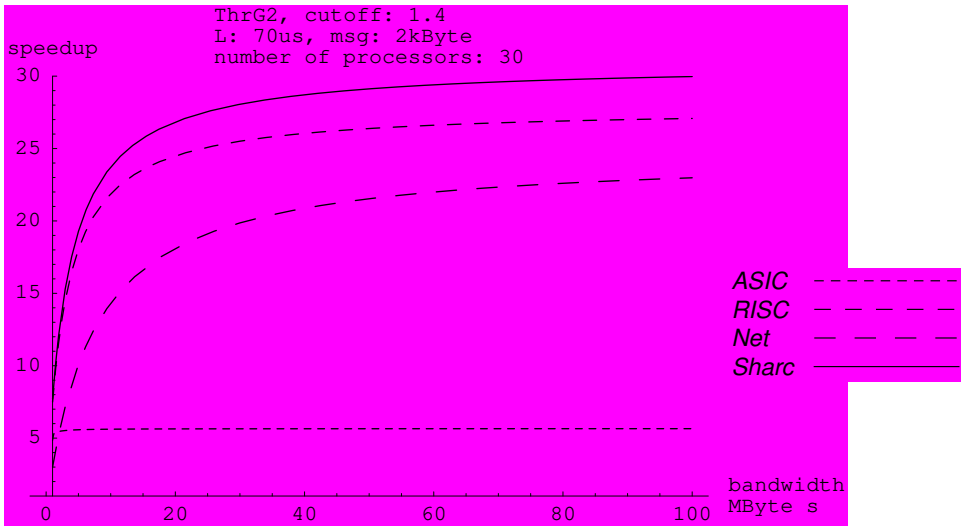
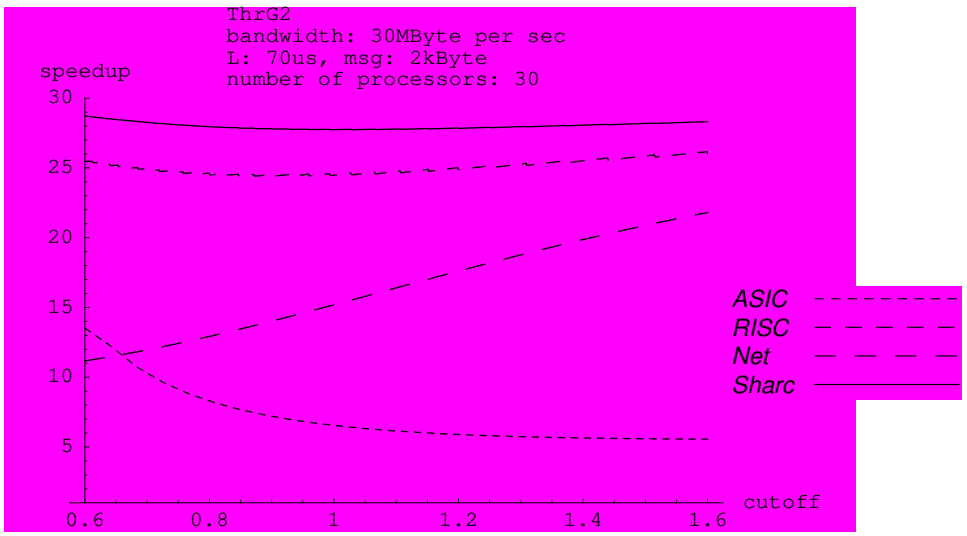


Figure 3.7 **a)** Estimated speed-up of the Net architecture with replicated data and point-to-point communication. Standard communication parameters are assumed for the PCI bus (30MB/s, $70\mu s$, 2kB). The (Myrinet) network parameters are varied as indicated.

b) Speed-up estimation, dependent on the bandwidth of PCI and *Net* network. Indicated values for latency (L) and message size (msg) apply for the PCI bus and the *Net* network; the number of processors is valid for all architectures. *Net* uses broadcast and collective communication. IBM SP-2 parameters are used for broadcast/collective communication.

The mapping of functions as well as the execution sequence of parallel function nodes are the same for the Net architecture and the RISC architecture. These architectures differ in the communication network. The PCI slots of the Net host machine do not contain coprocessor boards, but network interface cards to connect client computers. With eight-port full-crossbar Myrinet switches a workstation cluster with up to 48 nodes may be realised. The replicated data method is

a)



b)

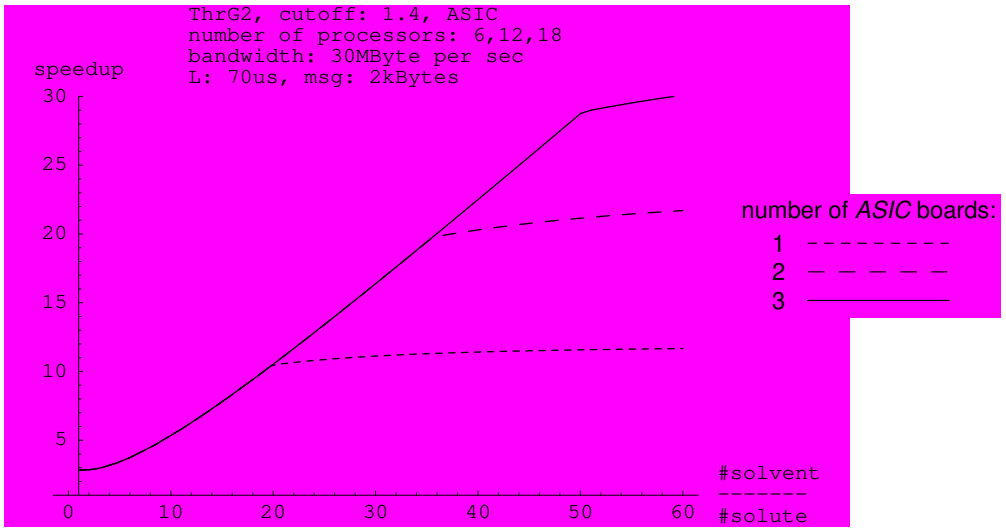


Figure 3.8 **a)** Speed-up estimation in dependence upon the cutoff radius. Indicated are values for bandwidth, latency (L) and message size (msg), applying for the PCI bus and the *Net* network. *Net* uses broadcast and collective communication with IBM SP-2 parameters.

b) Estimated speed-up of the ASIC architecture, in dependence upon the solvent to solute ratio (atoms). For the *ThrG2* benchmark, the ratio is about ten.

used to distribute particle coordinates as opposed to the RISC architecture. This is tenable because the clients are supposed to have enough memory. Due to the fact that Myrinet networks are switched, broadcast does not exist. If a parallel application program uses MPI broadcast or MPI collective communications, successive point-to-point connections are established on the Myrinet's physical layer.

Of course, the Net architecture is not limited to Myrinet networks and the Mathematica model distinguishes two options as well: point-to-point connections and broadcast. Point-to-point communication is easily implemented in the model by repetitively executing the same transmission. The aggregated point-to-point communication time is down-scaled if broadcast is enabled. The linear scaling function is determined by the achieved speed-up in the case of broadcast instead of point-to-point communication.

The Net architecture model is also applied to the IBM SP-2 parallel machine (appendix B). The SP-2 nodes are interconnected with each other through a multi-stage crossbar switch. Despite the switched network, broadcast communication is preferable if more than about four SP-2 nodes are allocated. An IBM SP-2 broadcast communication including 32 nodes achieves a speed-up of about four, compared with 31 successive point-to-point communications [21]. The same speed-up is expected for collective communications. The speed-up factor may be explained with the architecture of the SP-2 nodes: each node has four communication interfaces, thus is able to establish four switched connections concurrently [53].

The dependence graph of fig. 3.1b was used to map the MD algorithm onto the hierarchical Sharc architecture. Redundant nodes are mapped to intermediate-level Sharc processors and model the partial sums calculation (functional nodes *f9*, *f10*, *join*), as well as upwards and downwards communications in the hierarchy (fig. 3.2). Data distribution and gathering to/from Sharc boards is modelled the same as for the RISC and ASIC architectures. Additional communication cost due to the hierarchy is considered. While data is distributed, each Sharc processor stores the coordinates of particles residing in the associated slice and in neighbouring slices. This decomposition scheme is a trade-off between domain decomposition and replicated data. Communication during pairlist and forces calculation is not necessary. The add operation—required for partial sums on *f9*, *f10*, *join*—is modelled with 20ns per floating point addition.

The results of the presented performance estimations and the limitations of the discussed architectures may be summarized as follows:

- ASIC: the *solvSolv* performance of this architecture is the best of all (fig. 3.6a). The speed-up is limited either by pairlist calculations or *soluForce*, all of which not parallelised. In the presented estimations for *ThrG2*, *soluForce* is the limiting factor (sharp bend in fig. 3.6a). The ASIC architecture may reach a speed-up of ten only if 1) the pairlist for solvent is calculated on the coprocessor board, the pairlist for solute on host and 2) the molecular system contains at least 20 times more solvent atoms than solute atoms, at a cutoff of 1.4nm (fig. 3.8b).
- RISC: the RISC architecture is primarily limited due to the interprocessor communication during calculation, and secondarily due to the communication between host and coprocessor boards. The more coprocessor boards the system contains, the more point-to-point communications are required to gather partial forces from each board. This effect introduces the short but steep falls in the speed-up graphs (fig. 3.6b).
- *Net*: the Net solution is limited because all computational nodes (not only boards) communicate directly with the host. The reachable speed-up is strongly dependent on the quality of broadcast and collective communications the network provides. Scalability is rather bad if

only point-to-point communication is supported (fig. 3.7a). The most important fact is that a high bandwidth and low latency does not solve the problem: the *Net* solution of fig. 3.6a reaches the better maximum speed-up with moderate bandwidth and rather long latency.

- Sharc: the Sharc architecture is faster than the RISC architecture for two reasons. First, the performance of a single processing chip is better by a factor of 1.32. Second, no interprocessor communications occur during calculation, which is the primarily limitation of the RISC architecture. In addition, Sharc DSPs offer sophisticated communication capabilities (fast link ports with low overhead and DMA). The model shows that the gain of fast interprocessor communication exceeds the disadvantage of the hierarchy.

3.4.3 Assessment and Design Decision

Despite the fact that Sharc and RISC architectures are completely different, their performance behaviour is almost identical. If Sharc and RISC processors would be chosen to have the same performance, and if the same decomposition strategy applies, the RISC architecture would even outperform the Sharc architecture (with the same number of computational nodes). Therefore, in principle, not two different architectures are differentiated, but two decomposition schemes. One should keep in mind that complete implementation proposals (including the decomposition strategy) are assessed. The following assessment refers to the Gromos96P design goals as stated in section 1.5.2.

The Main Goal: Speed-up and Cost

The ASIC solution may not fulfil the performance requirement considering representative molecular systems with solute. Because the speed-up goal is the most important, the ASIC solution is out of the contest. All other proposals achieve the required speed-up of at least ten, but not necessarily with a total cost of less than 20,000 USD. This has shown to be difficult, since the RISC solution would require two boards with 10-11 processors, the Sharc architecture would require at least one board with a total of 13 processors. Both solutions might be too expensive; anyhow, the additional cost is expected to be within tolerable bounds.

The Net solution undoubtedly is too expensive: at least 10 client workstations are required. Together with Myrinet network interface cards and switches the cost constraint is exceeded. However, it is not mandatory to buy everything. If there are already enough workstations available, purchasing additional machines is superfluous. Moreover, the Net solution also runs on all networks, if the appropriate message passing library is available. As a conclusion, the Net architecture may be adapted to an existing workstation cluster without any additional cost.

Scalability

All examined systems scale very good for the required speed-up (at least ten). If the number of computational nodes is increased further, the scalability depends on many factors, e.g. the cutoff radius influences the ratio of the number of solvent to solute interactions. The presented estimations are based on the optimistic cutoff of 1.4nm. This is optimistic because a long cutoff increases the total number of interaction calculations per time step, and all parallel algorithms scale better if computation dominates communication.

Highest computational performance was assumed for RISC and Sharc architectures, presuming manually optimised code or even hand-coded assembler. In contrast, the code running on *Net* clients was not expected to be especially optimised. The assumptions of the Net network communication parameters are conservative as well: it is assumed that the message passing library does not employ a dedicated Myrinet driver but utilises protocol stacks instead.

Platform Independence, Maintainability and Development Cost

All parallel architectures require the parallelisation of the simulation software. Different software levels are separated, namely the application (GROMOS), device drivers (Myrinet, dedicated hardware boards) and firmware (the software running on coprocessor boards). The new parallel application is implemented completely platform independent, a standard message passing system has been used (MPI). To access possibly attached MD hardware, a device driver would intercept the interprocessor communication function calls of the application program. Device drivers are platform dependent by definition, whereas the firmware probably will not change for different platforms. However, if one of the functions, which is performed on the dedicated hardware, is subject to change, the firmware is also affected. This is a remarkable restriction in the flexibility of specialised hardware: changes of the MD algorithm are hard to employ. True platform independence may only be achieved with the Net architecture.

Implementing GROMOS96P on a Net architecture undoubtedly leads to the solution with the lowest development cost: as soon the parallel application is available, the Net solution is completed whether or not dedicated hardware and software is still in development. Development cost is essential for dedicated hardware. The total cost doubles if a dedicated hardware including appropriate low level software (device driver and firmware) is to be implemented, compared with the development cost of the parallel application based on message passing (no custom hardware).

The maintenance cost depends on the software structure and code organisation of the parallel application. Implementing the programming environment which has been introduced in section 2.4.4 is a promising approach to improve the maintainability and the ease of use of Gromos96P. Overall maintenance cost will not rise much if dedicated hardware is attached to the system. Once implemented and tested, firmware and device driver are not subject to change. Of course, each supported platform is requiring its own device driver.

Design Decision

Reviewing the given facts, the focus of interest is to develop a parallel GROMOS96P software following the concept of section 2.4.4, regardless of possible underlying hardware solutions. Following the trends in high performance computing, all evaluated architectures are distributed memory parallel machines with message passing communication. For our purposes, the MPI [21] message passing standard seems very suitable. MPI libraries are available for a wide range of platforms and operating systems. A great store was set on the declaration of the communication interfaces, such that possible future developments (coprocessor boards) are easily attached. In order to promote the development of customised device drivers, only synchronous point-to-point communication has been used in the prototype Gromos96P implementation.

Studying existing parallel MD software has shown that the replicated data method is the most simple parallelisation strategy. Consequently, GROMOS96P is based on a spatial decomposition of the simulation space, where the data is replicated on each computational node. An MPI compliant

message passing library is to be used for interprocessor communication. The fact that an MPI implementation must be supported on the target machine(s) is the only condition in order to preserve platform independence.

The next chapter deals with the parallelisation of GROMOS. Different decomposition strategies are compared, of which the spatial decomposition is employed using the replicated data method. Chapter 5 refines the parallel programming model which has been introduced in section 2.4.4. A graphical specification method for replicated data parallel algorithms is introduced together with the appropriate code generator.

3.5 Conclusions

The benefit of EA-based *system synthesis* to find suitable implementations for a given parallel algorithm strongly depends on the decomposition scheme of that algorithm. Classical Molecular Dynamics algorithms are data parallel algorithms, viz the same operation (e.g. interaction calculation) is repetitively executed on a given data set, whereas task level concurrency is of minor importance. However, the EA is best suited to find many design alternatives if the algorithm specification provides a sufficient amount of task level concurrency. Applying system synthesis to the MD algorithm has shown that only with some cunning techniques new implementation proposals were found. For example, artificial degrees of freedom have been added to the specification graph. Finally, the EA is only capable to find implementations for a specification graph with fixed execution times. In order to consider different parameter sets of the parallel algorithm or the architecture model, several specification graphs would be required and the synthesis repeated.

Analytical *performance estimation* of the most promising approaches was indispensable for two reasons: first, the synthesizer is unable to consider control flow, second, the need to predict the behaviour of possible target architectures for different molecular systems. Generic functional models incorporate Molecular Dynamics simulation parameters as long as they have a considerable relevance to the execution time of a function. In addition, basic control flow is considered, e.g. whether the pairlist is to be reconstructed or not. Speed-up predictions based on analytical models of architectures and the MD algorithm allow the designer to quickly assess design alternatives for different scenarios. A specific scenario is not only determined by Molecular Dynamics simulation parameters, but also by the chosen hardware composition like the number of processors or the communication bandwidth.

Generic models of Gromos functions and architectures were implemented in Mathematica [24] to estimate the performance of possible target architectures. The comparison of the results led to the *design decision*: A distributed memory environment based on message passing communication (workstation cluster with high speed network or general purpose supercomputer) is expected to fulfil our requirements.

CHAPTER 4

The Parallel GROMOS Algorithm

Exploring the design space in order to find a suitable architecture for parallel Molecular Dynamics algorithms has brought about the design decision that a workstation cluster with a fast interconnection network or a general purpose distributed memory supercomputer would be an adequate platform for a parallel GROMOS. General purpose architectures are not restricted to particular parallelisation techniques but allow the application to employ different parallel programming models. For that reason, commonly used Molecular Dynamics decomposition methods are investigated. Several factors like implementation costs, communication requirements, scalability and algorithmic overheads are taken into account to assess different decomposition methods.

One of the key issues of parallel MD algorithms is the distributed construction of the pairlist. This chapter starts with an overview of well-known efficient pairlist algorithms since similar principles are applied to the novel GROMOS96P algorithm. The already mentioned study about general MD decomposition methods follows. In section 4.3 it is shown how the domain decomposition and replicated data method are adapted to the GROMOS algorithm which is capable to handle truncated octahedron periodic boundaries as well. The applicability of these approaches is tested taking into account special algorithmic features like the twin-range pairlist method, the octahedron simulation box, and the calculation of the molecular Virial. Section 4.5 summarizes the results.

4.1 Neighbour searching

For systems consisting of a small amount of particles N, approximately less than 100, it is convenient to examine the interaction of all of the N(N-1)/2 particle pairs. This brute force approach requires too much computation time for systems of reasonable size. For this reason, Verlet[1] introduced a *neighbour table* in 1967, in which those particle pairs which are separated by a distance of less than $r_{ic} + r_s$ are stored (fig. 4.1a), where r_{ic} is the cutoff radius for which the interactions are calculated, and r_s is a small skin. The Verlet table led to a significant reduction in computation time because the search over all pairs of particles, which still scales with N(N-1)/2, needs only to be done at intervals. Between the updates of the neighbour list—also called pairlist—the program just has to check the particle pairs which appear in the list.

1. Verlet's original paper, dated 1967, is not referenced here, because the Verlet neighbour table format and update method is also well documented in [1] and [13].

The interval between pairlist updates is often fixed at the beginning of the run. The choice of the interval size depends on the thermodynamic state of the system, the length of the time step, and the values of r_{ic} and r_s. However, the update interval can be calculated dynamically at runtime by monitoring the displacement of each particle since the last update of the table. The basic idea is to choose the skin r_s thick enough so that a particle, which was outside (r_{ic}+r_s) at pairlist construction time, may not penetrate the skin in the time between two pairlist updates. Or, if the sum of the magnitudes of the two largest displacements exceeds r_s, the pairlist is reconstructed. Using this sum as an update criterion minimises the number of updates, but if the skin chosen is too thick and the update interval grows, this might result in an unacceptable drift in the energy [38]. Therefore, alternative displacement formalisms have been proposed [1][38].

While the Verlet neighbour table is constructed, all distinct particle pairs ij with a distance r_{ij} smaller than (r_{ic}+r_s) are considered. Every time step, the interaction is calculated between particles if $r_{ij} \leq r_{ic}$, in accordance with the neighbour table. If one of the particles currently reside within the skin area, the interaction is disregarded.

The calculation and bookkeeping of the displacement for each particle requires considerable time and memory, hence GROMOS makes use of a constant pairlist update interval which is determined at startup; on the other hand, the use of a brute force neighbour searching in GROMOS96 has a more harmful effect on execution time. Additionally, GROMOS employs a charge group neighbour list, as opposed to the original Verlet list which relates atom pairs.

4.1.1 The Link Cell Method

For molecular systems with thousands of particles, searching over *all* pairs to determine those fulfilling the cutoff criterion is too expensive, even if the pairlist is not updated every simulation step. Instead, an alternative method, the so-called cell index or Link Cell [1][13][64][38] method is more effective. Fig. 4.1 shows a two dimensional slice of a three dimensional (cubic) simulation box, broken into a regular lattice of cubic *cells* with an edge length h.

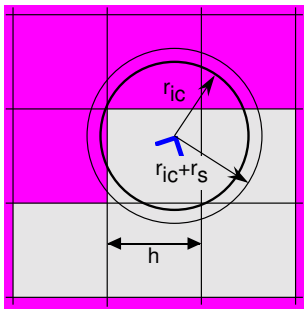


Figure 4.1 Link Cell method in two dimensions, showing nine square cells of width h, $h > r_{ic} + r_s$, as a part of the simulation box, with r_{ic} the cutoff radius for which the interactions are calculated, and r_s a small skin.

Because of the condition on the edge length, $h > r_{ic} + r_s$, the total number of cells depends on r_{ic} and r_s. If the edge length of each cell exceeds r_{ic}, then all interacting particle pairs ij are located within the same cell or in one of the eight neighbouring cells (26 in three dimensions). Using Newton's Third Law of pairwise forces, only the same cell and four neighbouring cells

(shadowed in fig. 4.1) have to be searched for neighbours (13 neighbouring cells in three dimensions). It should be noticed that early MD implementations in the seventies included all particles of the neighbouring cells in the pairlist *and* interaction calculation, that way introducing an artificial cubic symmetry which was called the cube-corner effect [64]. These outdated methods sometimes were also referred to as cell methods.

To determine the cell dimensions (h_x, h_y, h_z) and the number of cells, a three dimensional *rectangular* simulation box with edge lengths b_x, b_y, b_z is assumed. Accordingly, the number of cells along the axes are to be nb_x, nb_y, nb_z, see equation (4.1). The function '$\lfloor \ldots \rfloor$' accepts a floating point number as input and delivers the largest integer not greater than the argument.

$$nb_{x/y/z} = \left\lfloor \frac{b_{x/y/z}}{r_{ic} + r_s} \right\rfloor \tag{4.1}$$

The spatial extent of the cells is easily obtained by dividing the simulation box dimensions by the number of cells. Due to the nearest image convention, the number of cells must be at least two in each direction, hence, the cutoff radius may not be greater than half the smallest simulation box width.

All particles in the system are identified with an unique sequence number starting from zero up to N-1. The same is true for the cells, see fig. 4.2. The cell structure may be set up and used by the method of *linked lists* [1][13]. The usage of linked lists is illustrated in fig. 4.2 for the cells with sequence number zero and one. The example shows cell zero containing six particles and cell one with five particles. The particles are sorted spatially and accommodated in the arrays 'head' and 'chain' as follows: The index of the head array is the cell number. The value (e.g. head[1] = 6 for cell one) is the sequence number of the first particle of the respective cell. This sequence number serves as index in the chain array, which delivers the sequence number of the second particle of the cell (8). Again, the sequence number is used as index. The last particle of a cell is identified with the value -1 in the chain list.

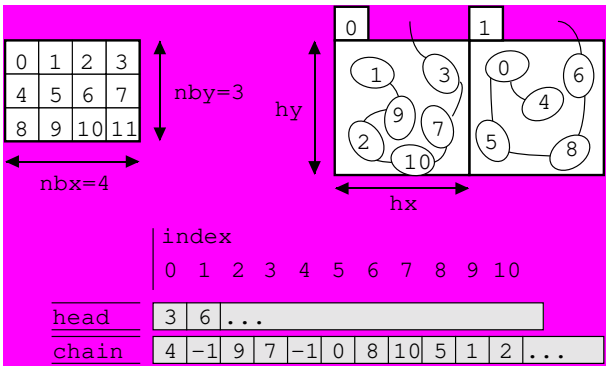


Figure 4.2 Cell method in two dimensions with a rectangular simulation box consisting of 4x3 cells. Note the cell numbering scheme in the upper left of the figure. A close-up of cell zero and one is shown in the upper right. The interconnection of the particles inside the cells have no physical meaning but indicate the sequence they will be processed. How the particles are stored in the arrays 'head' and 'chain' is shown at the bottom of the figure.

Using the linked list structures is very convenient in all cases where memory is an issue (e.g. when a system contains a large amount of particles), the main advantage being that the memory used always equals the number of particles plus the number of cells.

Assuming a homogeneous liquid, the number of particles per cell N_c is roughly constant. In order to construct the pairlist, the Link Cell method needs to check $(13.5 \cdot N_c \cdot N)$ particle pairs against the cutoff radius, whereas $(N \cdot (N-1)/2)$ operations are required for the brute force approach. If the cutoff radius increases to meet condition (4.2), the simulation box may be divided up in only four cells. In this case, the cell method coincides with the brute force search.

$$3 \cdot r_c > \mathrm{Min}[b_x, b_y, b_z] \tag{4.2}$$

In practice, MD simulations with a solvent are always designed to have minimum size, thus a typical cutoff should be expected to be relatively large compared with the simulation box dimensions. As indicated above, the efficiency of the cell method is not satisfactory with only a few cells due to two reasons: firstly, the search space grows as the cutoff radius grows, secondly, the examined volume per particle is at least $81/(4\pi) \approx 6.4$ times bigger than the cutoff sphere—*at least* because it was assumed that $r_c = h$ in the cubic case.

4.1.2 Grid Cell vs. Link Cell

Using a finer grid alleviates the influence of the cutoff, and the search space may be better approximated towards the cutoff sphere. The *grid cell* technique also uses a regular lattice of cells, but here the cells are chosen so small that not more than one particle occurs in one cell [64][38]. This corresponds to the situation where the volume diagonal of the cells is shorter than the smallest possible distance between two particles. All particles are spatially sorted into cells, e.g. by using a three dimensional array representing the grid. This array contains either a particle number or -1 if the cell is empty. A linked list is obsolete, because no cell is allowed to contain more than one particle.

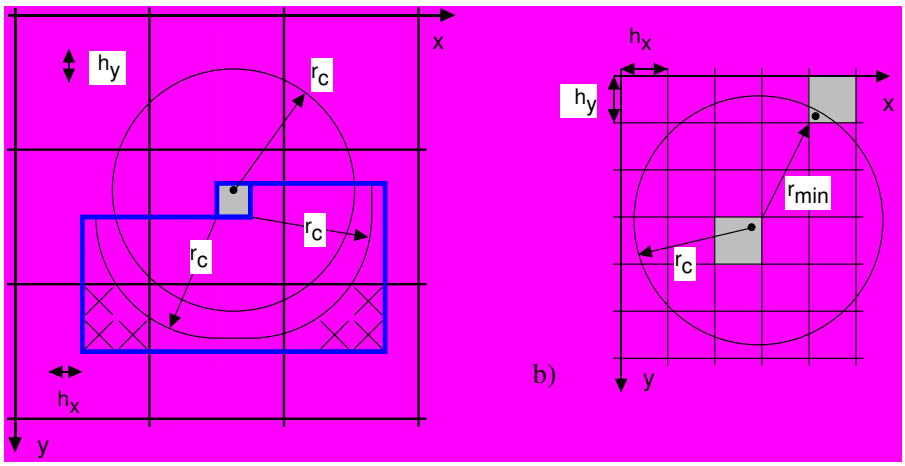


Figure 4.3 Grid cell method, illustrated in two dimensions. The interaction cutoff radius r_c indicates that no skin r_s is applied. Thus, r_c always refers to a static pairlist update interval.
a) Arbitrary cell grid, such that one cell contains at most one particle.
b) Calculating the minimal distance r_{min} between sub-cells.

The major problem of the grid cell technique is to find the neighbouring cells: given a non-empty cell somewhere in the lattice (e.g. the shadowed cell in fig. 4.3a) with cell coordinates (x_1, y_1, z_1). The cell coordinates are used as indices in the mentioned three dimensional array. Possible cell neighbours (x_2, y_2, z_2)—indicated with the thick line in fig. 4.3a—are easily found by accessing the array with indices relative to (x_1, y_1, z_1) as follows:

$$x_2, y_2, z_2 = (x_1 - \mathrm{ncpc}_x)\ldots(x_1 + \mathrm{ncpc}_x), (y_1)\ldots(y_1 + \mathrm{ncpc}_y), (z_1 - \mathrm{ncpc}_z)\ldots(z_1 + \mathrm{ncpc}_z) \tag{4.3}$$

With $\mathrm{ncpc}_{x/y/z}$ equal to the number of cells per cutoff and $h_{x/y/z}$ being the cell dimensions:

$$\mathrm{ncpc}_{x/y/z} = \left\lceil \frac{r_c}{h_{x/y/z}} \right\rceil \tag{4.4}$$

The function '$\lceil \ldots \rceil$' accepts a floating point number as input and delivers the smallest integer not less than the argument. Accessing the array with indices according to (4.3) delivers the cells indicated with a thick line plus four spare cells completing the rectangle. These spare cells must be eliminated. This is done by calculating the cell sequence number of the neighbouring cells (4.6). All neighbour-cells with a smaller index number than the primary cell are rejected.

Not all cell pairs according to (4.3) must be examined to find neighbouring particles. Because the cutoff sphere is better approximated with a finer grid, the cells identified with a cross are always too far away from (x_1, y_1, z_1) and their minimum distance exceeds the cutoff. How these cell pairs may be eliminated is sketched in fig. 4.3b, given the minimum distance r_{min} as

$$r_{min} = \sqrt{(b_x \cdot \mathrm{Max}[|dx| - 1, 0])^2 + (b_y \cdot \mathrm{Max}[|dy| - 1, 0])^2 + (b_z \cdot \mathrm{Max}[|dz| - 1, 0])^2} \tag{4.5}$$

with (dx, dy, dz) the integer connecting vector of the two cells under consideration. Two examples clarify how equation (4.5) is used:

1. Consider the two dimensional example of fig. 4.3b with $h_x = h_y = 1$, $r_c = 2.6$, $dx = 2$ and $dy = 3$. Equation (4.5) results $r_{min} = \sqrt{5}$, with $\sqrt{5} < r_c$. Thus, the distance of the particle pair (fig. 4.3b) must be checked against the cutoff.
2. The example in fig. 4.3a refers to $h_x = h_y = 1$ and $r_c = 3.6$, thus the number of cells per cutoff according to (4.4) is $\mathrm{ncpc}_{x/y/z} = 4$. The distances between the cells with a cross and the primary cell according to (4.5) are $\sqrt{3^2 + 3^2} = 4.243$ and $\sqrt{2^2 + 3^2} = 3.606$, respectively.

The grid cell algorithm approximates the cutoff sphere very accurately, hence the number of examined particle pairs is very close to the number of pairlist entries. The drawback is, of course, that many cells are empty.

It is shown in [38] that for a mono-atomic Lennard-Jones liquid near its triple point the cell size must be chosen so that about 90% of the cells are empty, as a result of which neighbour searching is not efficient. To allow a coarser grid, the particles may be redistributed to neighbouring empty cells not further away than a certain distance. Then, the search space is extended by this distance to also reach the displaced particles. Grid cell algorithms using this technique alleviate the problem of many empty cells, an efficient exemplary algorithm is presented in [38] and requires at most two cells per particle. A slightly different approach is applied to GROMOS96P.

4.1.3 GROMOS96P Pairlist Algorithm

In order to improve the performance of the existing pairlist algorithm its parallelisation was unavoidable. Without an efficient parallel pairlist implementation the major design goal, a speed-up of at least ten, could not be achieved (section 3.2.3). In connection with GROMOS, the new pairlist algorithm had to meet some preconditions:

- The cutoff radius r_c is specified at startup. Both pairlist update interval and r_c are constant, thus, there is no skin r_s.
- The algorithm must support rectangular, truncated octahedron, and monoclinic periodic boxes as well as nonperiodic systems.
- A parallel version of the algorithm for all supported types of boxes must be available.
- The compliance with the twin-range method should be maintained.

Simulations in nonperiodic systems permit the free choice of the shape of the grid. However, for periodic systems, the grid boundary must fit into the periodic box such that it also obeys the periodicity rules.

Rectangular Box

Similar to the Link Cell algorithm, the simulation box is partitioned into cells. To illustrate how periodic boundaries are considered, the edge lengths of the cells are chosen longer than the cutoff in fig. 4.4 which leads to $ncpc_{x/y/z} = 1$. Though, given a certain primary cell, only directly adjacent cells must be examined. The example simulation box is divided up into 4x3x5 cells, following the same cell numbering scheme as in fig. 4.2. The figure shows which cells must be searched for neighbouring particles under consideration of periodicity: for a particle residing in cell number six, possible interaction partners are in the same cell, or in one of the neighbouring cells as illustrated in fig. 4.4 (middle). For the primary cell 6, periodic boundary corrections never apply. This is in contrast to cell 11 (fig. 4.4, right), whose neighbouring cells also belong to periodically repeated images of the simulation box.

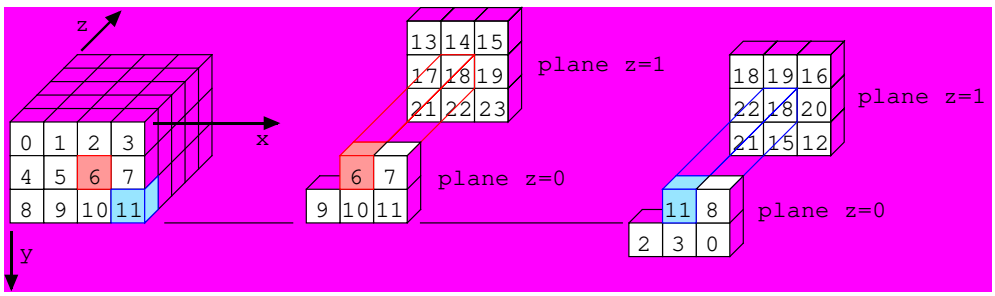


Figure 4.4 A three dimensional rectangular 4x3x5 simulation box (left) is shown, following the cell numbering scheme of (4.6) and assuming the number of cells per cutoff $ncpc = 1$. All particles fulfilling the cutoff criterion are in the same primary cell or in neighbouring cells as indicated for cell 6 (middle) and 11 (right). Note that for cell 11, periodically replicated images appear as neighbouring cells.

Given the cell coordinates (x,y,z) of any cell as integer values, the corresponding cell number is given by

$$\text{cellnumber}(x, y, z) = (y \cdot nb_x + x) + z \cdot nb_x + nb_y \tag{4.6}$$

Neighbouring cells are searched using (4.3) and (4.6) with periodic boundaries. First, the surplus cells are faded out: if the cell number of the secondary cell, derived from (4.6), is smaller than the cell number of the primary cell, the secondary cell is rejected. If the secondary cell is not rejected, the cell number must be recalculated in accordance with the periodicity rules. The components as a result of (4.3) must be corrected before they are fed to (4.6) as follows: if a component x/y/z is smaller than zero, $nb_{x/y/z}$ is added; else, the component is taken modulo the number of cells $nb_{x/y/z}$.

Because the Link Cell algorithm is not efficient with only $3 \cdot 4 \cdot 5 = 60$ cells, the number of cells is increased in practice. The total number of cells along the axes are chosen so that 1) the cells approximate a cube and 2) the total number of cells approaches the total number of particles. The cells are allowed to be empty or to contain more than one particle. As an additional constraint, the number of cells per cutoff $ncpc_{x/y/z}$ is the same in all spatial directions. This simplifies the search over neighbouring cells according to (4.3). The number of cells and their extent are given by an iterative algorithm in dependence with the number of processors (section 4.3).

The neighbour-cell search does not test minimal cell distances to determine if particle separations must be calculated at all. Instead, all neighbouring cells as delivered by (4.3) are evaluated. This means that cells with a cross (fig. 4.3a) are also considered. Experiments have shown that there is no performance gain by calculating the integer distance between cells instead of floating point distances between particles.

The GROMOS96P algorithm does not restrict the number of particles per cell. Therefore, to sort the particles spatially into cells, the implemented algorithm uses a linked list (fig. 4.2) as opposed to the grid cell algorithm which applies a three dimensional cell array for particle storage. The neighbour search uses cartesian coordinates to access the cells according to (4.3) and (4.6). For rectangular simulation boxes, the GROMOS96P pairlist algorithms may be summarized as follows:

1. Calculate the centres of geometries of solute charge groups according to (3.2). For solvent, the centre of geometry is the position of the first atom.
2. Calculate the number of cells $nb_{x/y/z}$ and the number of cells per cutoff ncpc such that the total number of cells tnbx approximates the total number of charge groups (section 4.3.1).
3. Sort particles into cells using a linked list (fig. 4.2).
4. Get a primary cell (sequence number and cell coordinates).
5. Determine all neighbouring (secondary) cells according to equations (4.3) and (4.6). Ignore cells with sequence numbers smaller or equal to the primary cell.
6. Search particle-particle pairs in the primary cell: if ncpc equals one or two, calculate all distinct particle pair separations. Check the distances against the cutoff and put the pairs into the list if necessary. If ncpc is greater than two, put all distinct particle pairs into the pairlist without evaluating the pair separation.

7. For each particle in the primary cell, calculate the distance to all particles in secondary cells, check the distances against the cutoff, and put the particle pairs into the list if necessary.

8. Get the next primary cell and repeat steps 5–7 until all cells are processed.

Truncated Octahedron

In GROMOS, the truncated octahedron is always the most regular one, that is a regular octahedron fitted into a cube with edge length b (fig. 4.5). In order to apply a grid to the truncated octahedron, two conditions must apply: 1) the grid must be chosen so that the truncated octahedron boundaries coincide with cell boundaries 2) the truncated octahedron boundary has to cut a cell into two equal parts such that the periodicity causes no overlap of grid cell contents at the truncated octahedron boundary. As a consequence of this symmetry requirement, the number of cells $nb_{x/y/z}$ along all axes must be four or an integer multiple of four.

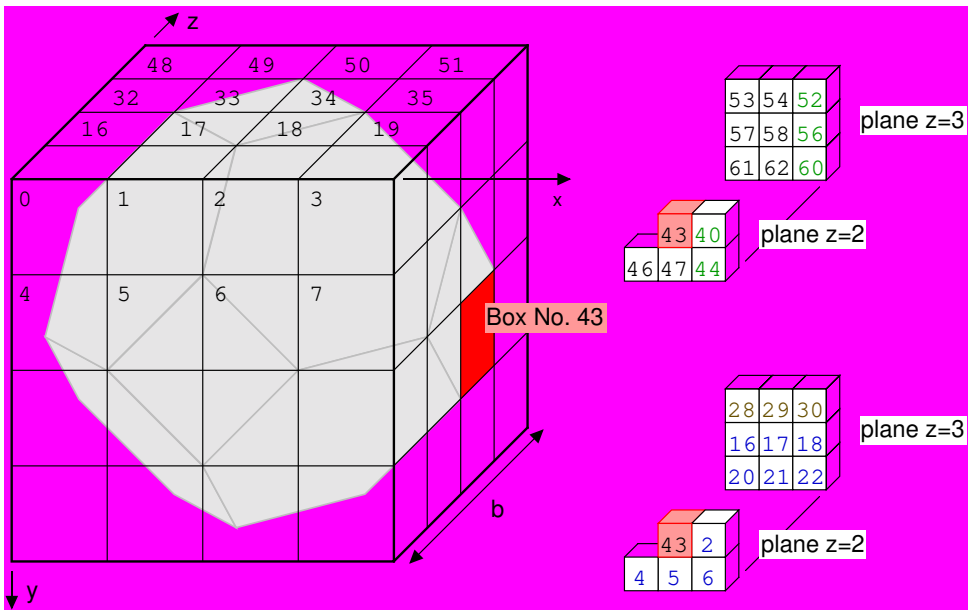


Figure 4.5 Truncated octahedron simulation box, fitted into a cube with edge length b, and with $ncpc_{x/y/z} = 1$: the diagrams on the right show how neighbouring cells of the primary cell 43 are determined in consideration of periodic boundaries.

Due to the nearest image convention, the maximum cutoff is $b\sqrt{3}/4$, which is half the length of the distance between two adjacent six-sided planes of the octahedron. If the cutoff is in the range between b/4 and $b\sqrt{3}/4$, the number of cells per cutoff ncpc will be two (with $nb = 4$). This means that—similar to the rectangular case—the neighbours are searched with the brute force method. To exploit the advantages of cell-based methods, the cutoff must be smaller than b/4, leading to $ncpc = 1$ with 16 cells. An example of an octahedron divided up into 4x4x4 cells is shown in fig. 4.5. The cell numbering scheme remains the same (rectangular box).

Finding neighbouring cells of a given primary cell is much more sophisticated with octahedron boundaries, as opposed to the rectangular case: fig. 4.5 illustrates that not the same neighbouring cells (cell numbers) are to be examined as for the rectangular case. Therefore, simply applying (4.3) as in the rectangular case would provide only a subset of the octahedron neighbouring cells, namely the cells depicted in the top right-hand corner of fig. 4.5. Neighbour cells in the plane $x = 4$ (outside the central box) come from the replicated octahedron adjacent to the four sided octahedron plane (depicted light grey in fig. 4.6). Additional neighbouring cells of cell 43 are also to be found in the replicated dark shaded octahedrons in the background. Neighbour-cells coming from these two octahedrons are depicted in the bottom right-hand corner in fig. 4.5.

A simple and straightforward algorithm to find these additional neighbouring cells could not be found. A brute force algorithm to find neighbouring cells has been reluctantly implemented instead. Each cell in the central box is spatially shifted (replicated) towards the primary cell by at most half the box length per coordinate component, at the same time minimising the length of the connecting vector. This way, only octahedrons adjacent to the six-sided planes are replicated. The minimum cell distance r_{min} is calculated (4.5) for all shifted cells and tested against the cutoff. The remaining part of the algorithm is identical to the rectangular one. Not surprisingly, the octahedron pairlist algorithm performs worse than its rectangular equivalent. Especially important is to find the best trade-off: with more cells fewer particle-particle distance evaluations (floating point operations) are necessary, at the expense of the brute force neighbour-cell search (integer distance calculation) which becomes the limiting factor. Experience has shown that there is no profit on modern workstations if floating point distance calculations between particles are replaced with integer distance calculation between cells.

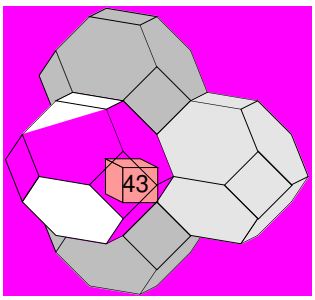


Figure 4.6 Replicated truncated octahedrons in which neighbour-cells of cell 43 may also be found (in addition to the central computational box).

Monoclinic box

The periodic monoclinic box is used for systems in the crystalline state, where the periodicity is determined by the crystal form. Within GROMOS, the oblique coordinate system (x', y', z') is defined as $y' = y$, with y' orthogonal to x' and z', and the angle β between x' and z'. With a simple coordinate transformation it is possible to adopt the oblique crystal system, so that the pairlist and the interaction functions designed for rectangular systems may be also be used for octahedron boundaries, with the minor change that the distance between particles must be corrected [5][26].

Performance

The performance of the new cell-based pairlist algorithms for rectangular and octahedron periodic boundaries is given in fig. 4.7, measured with a SunUltra30. Fig. 4.7a shows the speed-up of the new algorithms compared with the GROMOS96 pairlist function *nbnone*. If the cutoff radius is small compared with the simulation box extent, the advantage of the cell method is most apparent. The reason is because the examined volume with a small cutoff is significantly smaller than that with a long cutoff, whereas the brute force algorithm *nbnone* performs independently of the cutoff. The performance data of fig. 4.7 is determined with the *ThrG2* and *Thr2* examples, the simulation box length lies around 7nm for both benchmarks. Refer to table A.1 for exact values.

With a cutoff of 2nm, the cell-based octahedron pairlist algorithm does not seem to perform better than the brute force approach. The reason for this being that a 2nm cutoff is in the range between b/4 and $b\sqrt{3}/4$, and a search over all pairs is performed anyway. The bad performance of the octahedron pairlist is due to the expensive search for neighbouring cells. A trade-off between the number of cells and execution speed is hard to find, because the number of cells per axis may be increased only by multiples of four.

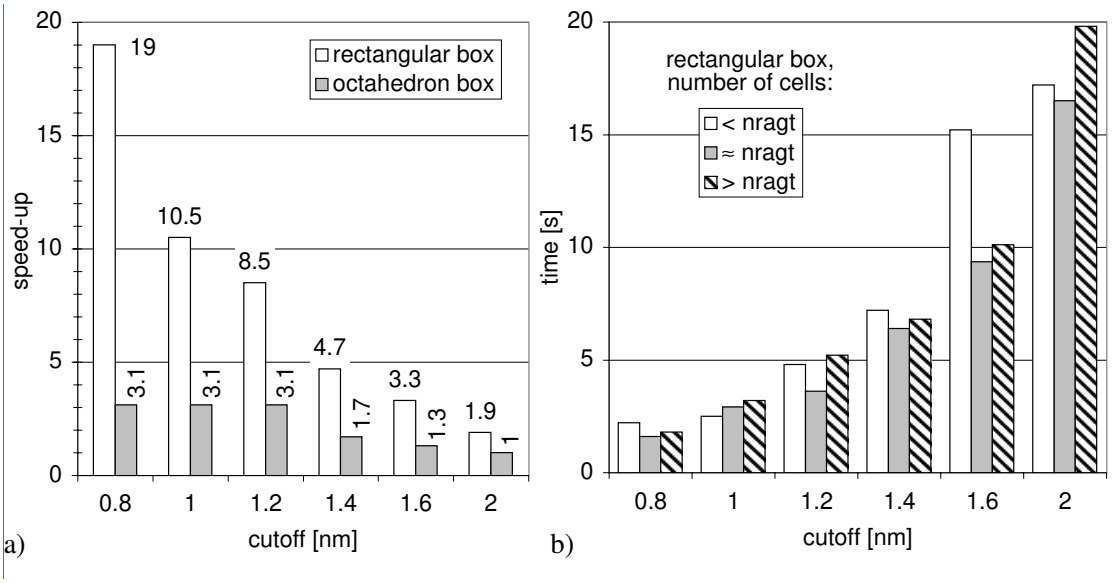


Figure 4.7 Performance of GROMOS96P cell-based pairlist algorithm.
a) Speed-up of the cell-based algorithm for both rectangular and octahedron periodic boundaries, with respect to the GROMOS96 brute force algorithm *nbnone*.
b) Performance of the pairlist algorithm with rectangular periodic boundaries is dependent on the total number of cells, with nragt = 12,246 the total number of particles (charge groups) in the system.

The pairlist algorithm for rectangular boxes is capable of processing an arbitrary number of cells. If the total number of cells approximates the number of particles (nragt), the algorithm reaches its maximum performance. The values in fig. 4.7b refer to the total number of cells relative to the total number of particles as follows, for different cutoff radii. The shadowed bar in the middle represents the best case execution time which is reached with one processor in the system,

and with the number of cells approximating nragt. Incrementing the parameter ncpc leads to many empty cells (hatched bar), decrementing ncpc results in many unnecessary particle distance calculations (white bar).

As a conclusion, the advantage of the cell-based pairlist algorithm is reduced with long cutoffs, regardless of the type of periodicity,

4.2 Decomposition Methods

Three decomposition methods are typically used with MD simulations. *Particle decomposition*, also referred to as atom decomposition, assigns unique particles to processors; the particles remain with the assigned processor throughout the simulation. *Spatial decomposition* [55] assigns partitions of the geometric space to processors; the particles may migrate to different processors based on their spatial coordinates. *Interaction decomposition* [62] assigns unique particle pairs to the different processors; the pairlists represent the interactions to be calculated during the simulation.

Interaction decomposition assigns all possible interactions to processors. Hence, no interprocessor communication is needed to calculate the forces. For the case that the number of processors is approximately equal to the number of particles, the interaction calculation remains local to each processor. Communication is required among processors that share common (primary) particles, which is the case only if there are more processors than particles. This method implies that pairwise identical forces are calculated twice. Interaction decomposition is not recommended for molecular systems with considerably more particles than processors.

The above mentioned methods are usually implemented with one of the following techniques: *systolic loops*, *replicated data*, and *domain decomposition*.

Systolic loop methods [29][58] distribute the atomic data equally over all processors and by passing data between them permit the calculation of all possible pair forces. For example, the coordinates are distributed over a ring of processors such that half of the coordinates passes every processor exactly once (mobile particles). Each node also stores the coordinates of a group of particles of the overall system (permanent particles). During a systolic cycle each processor evaluates and accumulates the interactions of the permanent particles with the passing mobile ones. Due to the geometric decomposition of permanent particles, generally it is not necessary to execute the complete systolic loop if a cutoff is applied. The computed forces are passed back over the ring to accumulate the global force on their residential processor. Particle decomposition usually is implemented with systolic loops.

The *replicated data decomposition* to Molecular Dynamics is a typical example of the data parallel approach which can be implemented for a large number of problems [54][47]. Each processor in a network has a copy of all atomic coordinates, velocities and forces and, in principle, attempts a full simulation of the system of interest, while sharing the workload equally with the other available processors. For inexpensive operations, for example a function with a complexity linear in the number of particles, there is little to be gained from parallelisation, so the computational effort is simply duplicated on each node. Time savings accrue where expensive program operations can be divided between the individual processors and carried out in parallel. However, some of the time saved by these operations is then negated by the need to circulate any changed

data around the network of nodes. The key to successful parallelisation rests with maximising the time spent in the parallel sections of the code and at the same time minimising the amount of data interchanged in the time consuming communication steps.

The *domain decomposition* [36][50] strategy is a well-known approach for the simulation of a large numbers of particles (multi-million systems) interacting with short-range forces. The GROMOS force field, for example, is such a short-range interaction, including Coulombic forces with the charge group concept. The simulation box as a whole is divided up into a series of *regions* of equal size, with each region handled by a separate computational node. The regions themselves are divided into cells of equal size with the number of cells maximised, subject to the condition that none of the cell sides are smaller than the cutoff radius. Particles in a particular cell will interact only with particles in the same or adjacent cells, as illustrated in fig. 4.4. Assuming the box of fig. 4.4 being a region on one computational node, special treatment is needed for cells on the boundary of that particular region. Message passing is used to communicate particle positions and interactions. That way, communication with three neighbouring regions or computational nodes is required. After all the forces have been calculated, the particle positions are updated. Since the domain decomposition has its basis on geometry, all the data structures must be updated to account for positional changes. The particle coordinates are checked and if a particle is in the wrong cell it is moved to the proper cell. If the new cell is on a different computational node, message passing is used to send the particle to the new computational node. Each node checks for incoming particles and places them in the proper cell when received. However, the amount of data required to be passed is much smaller than that needed for a global sum. This potentially means that this algorithm provides substantial time savings over the replicated approach. Communications due to the region overlap is minimised if the regions are cubes. Recapitulating the domain decomposition, a spatial decomposition without replicating the data has been described, where missing data in the overlap zone between the regions is replicated or communicated.

The choice of an algorithm depends on several factors: The replicated data method is convenient for a few hundred processors and up to about 50,000 atoms, since it is adapted to a parallel system by straightforward means. Domain decomposition algorithms are very effective for multi-million particles simulation [49], allowing dynamic load balancing for heterogeneous particle density. Domain decomposition is best suited to reduce the communication overhead which hampers the scalability of replicated data implementations. In return, the complicated communication scheme itself induces a considerable overhead, which is not dependent on the number of processors. To determine a possible trade-off, UHGROMOS (replicated data) and its further development EulerGromos, which employs the domain decomposition, are compared in [36]. The performance of both implementations have been measured on Myoglobin, a molecular system consisting of 10,914 atoms, with a cutoff radius of 1nm. The measurement was done on a 512-node Intel Touchstone Delta machine (32GFLOPS peak performance, 16MB memory per node) and it was pointed out that the replicated data version outperforms EulerGromos if less than about 60 computational nodes are allocated. For larger molecular systems or smaller cutoff radii the crucial advantage of domain decomposition will become evident for a smaller number of processors. This has been shown in section 3.4.2 (performance estimation).

4.3 GROMOS96P Spatial Decomposition

The ratio of the number of processors to the number of particles has proven to be the crucial factor in order to determine if domain decomposition outperforms the replicated data approach. Particularly, the replicated data approach will perform badly with a large number of processors and a moderate number of particles (less than 100,000 is considered as moderate). For that reason it was evident to contemplate a GROMOS96P implementation applying the domain decomposition scheme. At first sight, the implementation of an elaborated communication scheme seems to be the only difference between a domain decomposition and the replicated data alternative.

The domain decomposition algorithm as roughly described in the previous section assumes that no global communication is required. Remembering the GROMOS MD algorithm (section 1.4.2) and assuming that the functionality of GROMOS96P must not be reduced, global communication *is* required each time step. Depending on the selected features, scaling factors must be distributed to all computational nodes and/or solute particle positions must be replicated. The reasons are as follows:

- Temperature scaling is used to keep the total energy at a constant value, which is a required in closed systems. The unconstrained velocities (1.22) are scaled with the temperature scaling factor which is derived from the kinetic energy of all degrees of freedom in the molecular system and the reference temperature.
- In addition to temperature scaling, GROMOS provides pressure coupling to keep track of the constant total energy of the system. The Virial (1.20) is used for pressure coupling as opposed to the kinetic energy for temperature scaling.
- In most systems, the fast oscillations of covalent bonds are handled with the iterative SHAKE algorithm. Solvent molecules are never split over region boundaries and could be handled independently on each processor. Solute molecules consisting of several charge groups *are* split over regions. Moreover, solutes usually overlap regions by more than the cutoff radius depending on the extent of the molecule.
- A similar problem arises if the *molecular* Virial is to be calculated. To calculate the centres of mass positions for solutes (1.17), all solute coordinates are needed which are spread over the entire simulation box.

The first two points refer to the total kinetic energy and the *molecular* Virial, respectively, both of which not contributing to a serious amount to global communication. The two latter cases are worse, since all solute coordinates are needed. Nevertheless, a modified spatial decomposition scheme has been implemented.

4.3.1 Spatial Decomposition with a Rectangular Box

The rectangular simulation box is divided up into regions of equal size such that the number of regions match the number of computational nodes. The axis along which the system is decomposed is chosen to be the longest side length of the simulation box. Fig. 4.8 illustrates a spatial decomposition along the z-axis, since $b_z > b_x, b_y$ (a glossary concerning decomposition parameters is provided in section 4.5.2).

The number of cells are iteratively calculated as follows:

1. Set the initial number of cells along the z-axis to be equal to the number of processors. By this, the number of regions is determined:
   $nb_z = P_{tot}$; the initial number of cells along the z-axis.
2. Calculate the width of one cell. Within the first iteration, h_z is the width of the regions:
   $h_z = b_z / nb_z$
3. Calculate the number of cells per cutoff: ncpc.
   $ncpc = \lceil r_c / h_z \rceil$
4. Given the number of cells per cutoff, calculate the number of cells along the x- and y-axis:
   $nb_x = \lfloor (ncpc \cdot b_x)/r_c \rfloor$ and $nb_y = \lfloor (ncpc \cdot b_y)/r_c \rfloor$
   The number of cells along the secondary axes are not dependent on P_{tot}, that way approximating cubic cells.
5. Calculate the total number of cells: tnbx.
   $tnbx = nb_x \cdot nb_y \cdot nb_z$
6. If tnbx is smaller than the total number of particles (nragt), increase the number of cells along the z-axis $nb_z = nb_z + P_{tot}$ and repeat steps 2–5; otherwise, the number of cells along all axes is determined.

4.3.2 Domain Decomposition Algorithm

The GROMOS96P spatial domain decomposition is described next, in conjunction with the MD algorithm as introduced in section 1.4.2, assuming message passing communication. How particles are communicated is illustrated in fig. 4.8.

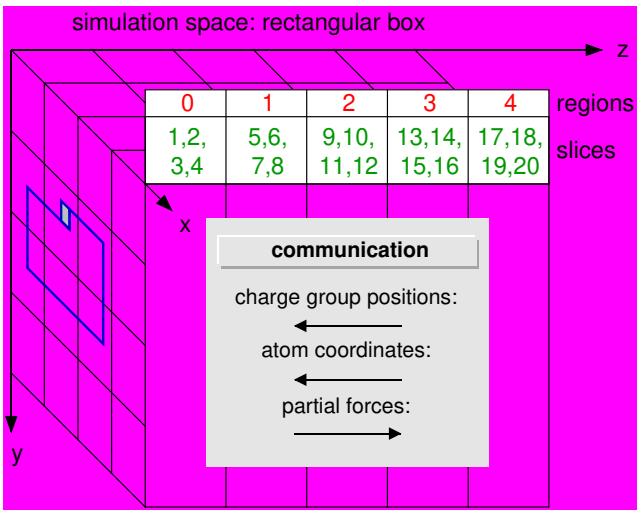


Figure 4.8 GROMOS96P spatial domain decomposition. The simulation box is partitioned into regions of equal size, the number of regions matching the number of processors. Each region is further subdivided into slices and cells in accordance with the pairlist algorithm.

Fig. 4.8 shows a rectangular simulation box decomposed along the z-axis. There are $P_{tot} = 5$ regions and $nb_z = 20$ slices along the z-axis, and the number of slices (or cells) per cutoff is $ncpc = 4$. To simplify the description of the basic steps of the algorithm, a unique number of slices N_s per region is assumed. This is not a requirement since nb_z / P_{tot} is not necessarily an integer number. The algorithm specification refers to those steps executed solely on the host machine with capital letters, whereas the numbers refer to tasks which are executed on all clients concurrently.

A If a pairlist is to be calculated in this step, the host machine sorts all charge groups spatially and assigns them to cells, slices and regions.

B If the molecular Virial is to be calculated, determine the atomic positions relative to the centre of mass for solute molecules (1.18).

C The host distributes all atom coordinates to client processors. Each processor stores only the coordinates of the particles which are currently associated with its region. In the case of Virial calculation, the relative atomic positions of the solute molecules are replicated on all regions.

1. In order to prepare the pairlist construction, calculate the charge group positions (centres of geometry) for each region. Alternatively, charge group positions are received from the host to avoid duplicate operations.

2. Determine the initial number of slices n_{s0} to communicate: $n_{s0} = |ncpc - N_s| + 1$
The initial number of slices are required to completely calculate the pairlist of those particles residing in the first slice of a region. Each region sends n_{s0} slices asynchronously to the -z neighbour node, starting with the lowest slice sequence number. From the +z neighbour node, n_{s0} slices are asynchronously received. Associated with the slices are charge group positions.

3. Start the pairlist iteration: loop over all slices $i = 0, \ldots, N_s - 1$ in the region.

4. The primary slice is selected: the sequence number of the primary slice is determined by the sequence number of the first slice in the region plus the slice counter i.

5. If $i = N_s - 1$ (last iteration), skip this step. Else, check if additional slices must be communicated for the next iteration. This is the case if $(ncpc - N_s + i + 1) > 0$. At most one slice is sent and received within the loop. The slice which is sent to the -z node is determined by the index number of the first slice of the region plus an offset. This offset is $(i + 1 + ncpc)$ modulo N_s, and is only valid if N_s is valid for all regions. The appropriate asynchronous send and receive operations are initiated.

6. Search charge group neighbours within the primary slice in accordance with the algorithm of section 4.1.3 for the rectangular box.

7. Check if the required subsequent (+z) ncpc slices are locally available. If not, terminate the corresponding outstanding asynchronous receive communications.

8. Search the neighbours for particles residing in the primary slice in ncpc subsequent slices.

9. Increase the loop (slice) counter i, check the loop termination criterion and proceed with step 6 if required.

10. Check if all asynchronous communications have terminated.

11. If required, prepare the Virial calculation for all solvent molecules residing in the region; the solute is handled on the host (step B of the algorithm).

12. The force calculation then proceeds using the local pairlist to evaluate forces only for those charge groups which properly reside on each node. The processing sequence of the cells is the same as for the pairlist construction. The same communication scheme applies to transfer non-resident atom coordinates to and from neighbouring nodes. Consequently, the forces must be transferred back to keep track of the pairwise identical interaction (fig. 4.8).

13. Each computational node sends atomic forces and partial energies and the Virial to the host.

D The host restores the global force array (no summation required) and sums up the energies and Virial.

E Additional operations on the host apply in accordance with the algorithm (bonded forces, integration, pressure and temperature scaling, file I/O, etc.)

F End of time step. If a pairlist is to be constructed in the next step, go to step A of the algorithm. If no pairlist will be calculated in the next step, the particles are not redistributed, thus, go to step B of the algorithm.

Considering the communication strategy, the presented algorithm is not a standard (ideal) domain decomposition for a number of reasons: 1) standard domain decomposition algorithms approximate cubic regions. With cubic regions instead of slices interprocessor communication is reduced. 2) After each time step, all forces are collected on the host, whereas only solute forces are absolutely required.

Compared with traditional domain decomposition approaches, the presented algorithm introduces additional communication cost: within each time step, solvent coordinates are distributed and solvent interactions are collected; in terms of GROMOS MD parameters, the additional amount of $(2 \cdot \mathrm{ndim} \cdot \mathrm{nvcag} \cdot \mathrm{nram})$ floating point numbers (table 3.4) are communicated. For the *ThrG2* benchmark (table A.1), this is 1MB of data (single precision floating point). In return, the presented concept offers a simplified communication scheme, since communication occurs only along one axis. Additionally communicating the solvent back to the host has a nice side-effect: most of the existing GROMOS code is not subject to modifications and is executed on host only, especially the most complicated functions *bonded*, *integration*, and SHAKE (fig. 3.1). The scalability is only slightly reduced if these function are not parallelised, since they are not very complex in terms of computation time. In order to estimate the performance characteristics, the domain decomposition algorithm has been employed for the *RISC architecture* (section 3.4.2).

The following modification of the presented algorithm would allow minimising the global communication with GROMOS: the functions *bonded* and *integration* (fig. 3.1) are also executed on the clients to evaluate the unconstrained velocities. Energy and Virial contributions are sent to the host for global summation. The host then calculates temperature and pressure scaling factors and broadcasts them to the clients. The clients apply the scaling factors to determine the constrained velocities and new positions. In order to SHAKE the solute, the unconstrained solute coordinates (derived from temperature scaled velocities) are required on the host. In addition, solute coordinates are required on the host to prepare the Virial for the next time step, see equations (1.17) and (1.18). Consequently, either the solute forces or the already integrated new unconstrained positions must be communicated to the host in order to SHAKE the solute and to prepare the Virial. Otherwise, constraints dynamics cannot be supported in a parallel version.

4.3.3 Replicated Data

Rectangular box

The replicated data decomposition broadcasts the atomic coordinates to all connected processing nodes, and each node stores the whole set of data. Replicated data parallel algorithms are most simple to program due to the very simple data distribution and gathering. Only minor changes are needed to parallelise existing sequential functions, because the same global data structure may be used. A replicated data decomposition is employed in the *Net* architecture (section 3.4.2). Again, functions like bonded forces calculation, integration, scaling, etc. are performed on the host machine. The prototype GROMOS96P implementation also employs this type of replicated data.

The first assumption that modern computer networks would provide an efficient broadcast communication was a fallacy. As mentioned earlier, Myrinet network and IBM's SP-2 network have been investigated. Both networks allow only point-to-point connections over cross-bar switches. Broadcast communications are translated by the message passing library into subsequent point-to-point communications. Thus, if the Myrinet is used to interconnect workstations, the predicted maximum performance is reached with about 15-25 clients (fig. 3.7a). The SP-2 interprocessor communication is capable to establish four connections simultaneously, thereby providing a limited low-level broadcast. The performance estimation for the SP-2 has shown that even with this limited broadcast a good scalability is maintained with up to 12-16 computational nodes, and maximum speed-ups of up to about 25-35 can be achieved (fig. 3.6b).

Point-to-point communication has proven to be the limiting factor if the number of processors exceed a certain number. How the total amount of communicated data could be reduced by solely using point-to-point connections is shortly outlined. The global data structure containing the particle coordinates is packaged into smaller units, each unit destined for a specific region. Such a unit of coordinate data defines a *completed region* containing the information about the particles actually residing in a particular region and, in addition, all positions of particles in the overlap zone. The overlap zone is determined by the cutoff radius and has a thickness of ncpc slices. The client processors receive their unit and restore the global data structure, replenishing the gaps with zeros. Gaps occur for non-resident and non-replicated particles. By distributing only *completed regions* only data which is actually needed is replicated on each node, as opposed to the fully replicated data approach, where all data is replicated (and communicated) on each node. For example, the *Sharc* architecture (fig. 3.2d) is supposed to replicate only the overlap zone on each DSP chip, thereby taking advantage of *completed regions*. The respective Mathematica model assumes a perfect broadcast to each board and hardware supported on-board data distribution. In contrast, the Mathematica model of the *Net* architecture assumes fully replicated data.

If domain decomposition techniques are applied, the overlap zone is communicated by using an elaborated communication scheme (section 4.3.2).

Octahedron box

Domain decomposition is exceedingly difficult to apply to the octahedron simulation space. Since there is already a significant overhead for the rectangular box, the gain with an octahedron box, if any, is assumed to be marginal. Besides, one should keep in mind that the octahedron is primarily used to minimise the number of particles in the simulated molecular system. Compared with the box extent, a long cutoff will be the regular case, for which no significant savings in communications may be achieved. Moreover, the domain decomposition communication scheme requires

data transfers (particle coordinates in the overlap zone, partial forces) to neighbouring cells which may not be determined in an efficient manner. The identification of neighbouring cells is also required to package the data into *completed regions*. Neighbour-cell searching is very expensive. If it is executed on the host only, its long sequential execution time limits the scalability of the parallel system. Thus neighbouring cells are searched on the clients only for regions. This must be done either each time step in case of domain decomposition or only if a new pairlist is to be calculated in case of replicated data. Consequently, the prototype GROMOS96P implementation decomposes the octahedron with a fully replicated data method.

4.3.4 Load Balancing

Decomposing the rectangular box assures that nb_z is an integer multiple of P_{tot}, resulting in P_{tot} regions with equal volumes. Consequently, load balancing is perfect for homogeneous liquids (fig. 4.9a/b). If $P_{tot}^3 \gg nragt$, the total number of cells tnbx will be much greater than the number of particles, and the pairlist algorithm slows down (fig. 4.7b), at least without any impact on the good load balancing.

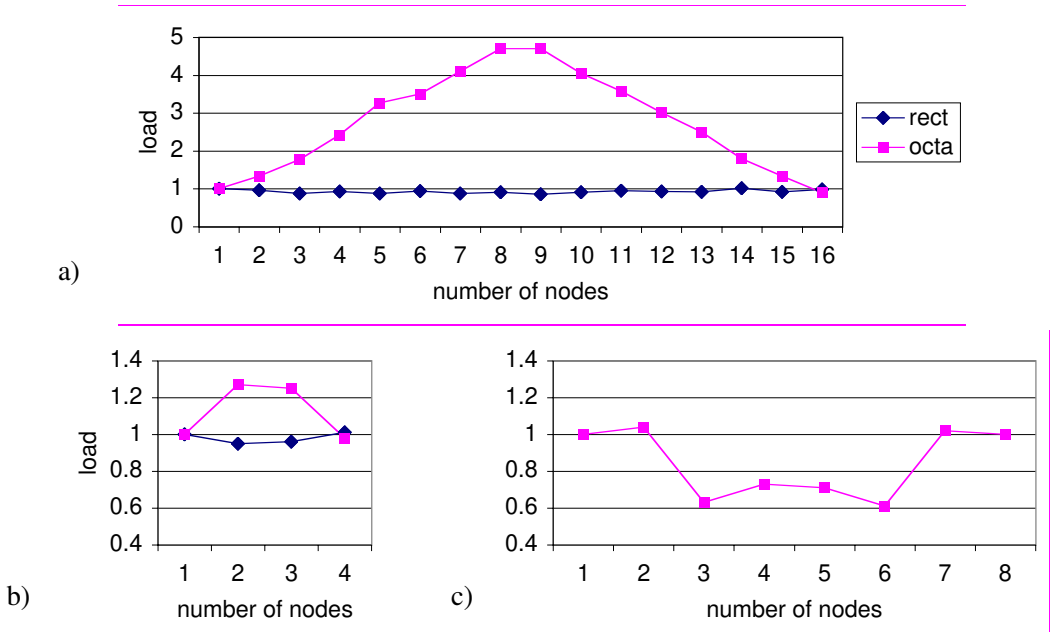


Figure 4.9 Load Balancing with fully replicated data decomposition. The values are normalised so that the host (node 1) has the load 1. The number of pair interactions served as a basis for assessment. All figures refer to the benchmarks *ThrG2* (rect) and *Thr2* (octa), both of which decomposed into $16 \cdot 16 \cdot 16$ cells to illustrate the effect of irregular slice grouping in the octahedron case. The cutoff is $r_c = 1.4\text{nm}$.
a) Good load balancing is assured for rectangular boxes, since the number of slices is always an integer multiple of P_{tot}. If the number of slices are equal to the number of nodes, irregular slice grouping is not possible to distribute the octahedron workload.
b) 16 slices are grouped to build four regions. A regular pattern (4,4,4,4) applied for the rectangular box, an irregular pattern (5,3,3,5) applied for the octahedron.
c) 16 slices are grouped into eight nodes following the pattern (4,2,1,1,1,1,2,4).

Using the same load balancing strategy for the truncated octahedron as for the rectangular box, namely dividing the enclosing cube into rectangular regions of equal volumes, would result in a heavily unbalanced workload: a central region would contain much more particles than a border region (fig. 4.5). Apparently, this problem is easily solved: first, the number of cells is increased. Then, the regions are compiled in a way that a border region consists of more slices than a central one. If the grid is chosen fine enough, the octahedron share to unique region volumes is well balanced out throughout all regions. Finer grids increase the total number of cells to exceed the number of particles, and the pairlist calculation slows down drastically (section 4.1.3). Considering the *Thr2* example of fig. 4.7a with a 1.4nm cutoff, the moderate speed-up of a factor 1.7 has been achieved with only $16 \cdot 16 \cdot 16$ cells. Decomposing the octahedron with 20 cells per axis increases the pairlist construction time to be worse than the brute force approach. Because the cell-based pairlist algorithm scales very well with the number of processors, this loss of performance at least scales down as well. A well-balanced workload can be achieved if $P_{tot} \ll nb_{x/y/z}$ (fig. 4.9).

Load balancing is not an issue in the case of rectangular boxes if molecular systems with solvent are being simulated. For octahedron boundaries, load balancing and scalability improves as the molecular system grows. As a rule of thumb, a good load balancing is assured if the number of processors does not exceed the ratio of the simulation box length to the cutoff radius. In the case when there are more processors, the current implementation selects the decomposition parameters in a way that the performance loss due to a workload imbalance is equalized with a faster pairlist construction.

In order to achieve a better load balancing for octahedron boxes, both pairlists (solute and solvent pairlist) may be split into P_{tot} equal parts (interaction decomposition). Parallelising the nonbonded computation with the interaction decomposition method divides the computational work perfectly, since all kinds of interaction calculations are of similar complexity. Interaction decomposition needs the pairlist to be calculated on host, thus it is not applicable to GROMOS96P (table 3.3).

4.4 Parallel Twin–Range

The twin-range pairlist method is a common method to handle long-range forces (section 1.3.3). In order to evaluate the nonbonded interaction with sufficient accuracy, a long cutoff radius r_c has to be used. For molecular systems, a value of at least 1.4nm seems necessary. If, for example, water is simulated using a pairlist with a 1.4nm cutoff, each water molecule will have about 200 interacting neighbours.

The nonbonded interactions are evaluated at every simulation step using the charge group pairlist that is generated with a short-range cutoff r_c. Longer range interactions between charge groups at a distance longer than r_c and smaller than r_{cl} are called *twin-range interactions*. Twin-range interactions are evaluated less frequently, advantageously when the pairlist is reconstructed. They are kept unchanged between these updates, thus, fluctuations of the forces beyond r_c are neglected during several steps.

Gromos evaluates the twin-range interactions using the reaction field corrected electrostatic term and normal Van der Waals parameters in the Lennard-Jones term (1.26). It is assumed that no excluded neighbours, no third-neighbour- or 1–4-interactions, and no intra-charge-group interactions exist at these distances. In order to accomplish that, r_c must not be chosen too small. The evaluation of nonbonded forces in the twin-range needs only a subset of the functions provided by *soluForce*. Therefore, twin-range calculations may be simplified in a similar manner as done in *solvSolv*.

There are basically two possibilities to determine twin-range interactions in a parallel system: 1) sequentially on the host and 2) in a parallelised and distributed manner like short-range nonbonded interactions. The first proposal would require the minimum number of changes in the existing GROMOS96P code. The already existing function *nbwith* (GROMOS96) determines twin-range interactions at the same time as the short-range pairlist is constructed. With minor changes *nbwith* would be reused to evaluate the twin-range on the host without generating a short-range pairlist. As it is the case for *nbnone*, *nbwith* applies a brute force neighbour search too. The short-range pairlist would be constructed as usual on all computational nodes. The major drawbacks of this solution are firstly a sequential twin-range calculation (scalability limitation), and secondly that charge group neighbour searching is performed twice. As an advantage, a shortened cutoff radius applies to the parallel part, leading to a faster pairlist construction, improved load balancing for octahedron boxes, and less memory demands.

The proposed parallel twin-range algorithm divides up the simulation space in accordance with the GROMOS96P spatial domain decomposition (replicated data) with the only difference being that the long-range cutoff is used. The parallel pairlist construction must be modified such that an additional pairlist including the twin-range particle pairs is generated. The additional pairlist does not separate solute and solvent interactions, thereby requiring a new parallel function to evaluate twin-range interactions, e.g. a modified *nbwith*-function. In addition, the functions to sum up partial forces and energies have to be adapted to incorporate the twin-range contributions. The pros and cons of the parallel twin-range are very different from those mentioned above for the sequential twin-range proposal. Currently, GROMOS96P deals twin-range interactions by increasing the short-range cutoff up to the value of the long-range cutoff radius.

4.5 Summary

4.5.1 Key Results

In this chapter a systematic comparison of commonly used decomposition methods for Molecular Dynamics algorithms has been carried out. The major results are:

- Competitive parallel MD implementations make use of spatial domain decomposition methods. Other decomposition schemes are scarcely used.
- *Domain decomposition* provides excellently scalable implementations especially well suited for large scale molecular systems. For systems of moderate size the overhead due to the very complex communication scheme becomes the dominant factor. As a rule of thumb, domain decomposition is the right choice if 1) more than about 50 computational nodes are used, and 2) only if the cutoff is significantly shorter than the simulation box extent.
- The *replicated data* method outperforms the domain decomposition scheme on parallel computers if only a small number of processors is used. This method is certainly the best choice if the cutoff is rather long compared with the simulation box extent. The scalability of a replicated data implementation is limited regardless of the communication subsystem chosen. Scalability may be improved using an efficient broadcast mechanism instead of point-to-point communications or by replicating not all data but only the spatial overlap zone which is defined by the cutoff.

The results of this comparative study affected the choice of the GROMOS parallelisation technique as follows:

- A typical GROMOS simulation consists of less than 50,000 particles. The resulting simulation box is not large enough in comparison with the cutoff; thus there is no profit in the domain decomposition method.
- Global communications additionally reduce the performance of a domain decomposition. GROMOS requires global communication in two cases: 1) if the molecular Virial is to be calculated and 2) if the simulation box is a truncated octahedron.
- As a novelty an algorithm enabling parallel simulation of octahedron periodic boundaries was developed. This algorithm assumes that particle coordinates are replicated on each computational node.

Consequently, GROMOS96P employs the replicated data method. The cell-based parallel pairlist algorithm divides the simulation space into *regions*. In case of a rectangular box, the regions have equal size, thus a good load balancing is ensured. However, coupling the load balancing to the spatial decomposition of the pairlist has two significant drawbacks: first, a good load balancing for *octahedron* boxes is difficult to achieve, second, the pairlist algorithm itself performs badly if the number of processors increase.

The undoubtedly major advantage of the employed replicated data method is that parallelising the entire GROMOS96 MD code is not required. Therefore, only the most computation time intensive parts of the algorithm are parallelised, whereas complicated tasks like integration and bonded forces calculation are executed by GROMOS96 routines on the host.

4.5.2 Decomposition Parameter Glossary

Parameter	Explanation
P_{tot}	The total number of computational nodes or processors in the parallel system.
$nb_{x/y/z}$	The number of cells along a selected axis.
$h_{x/y/z}$	The cell extent.
r_c	Regular interaction cutoff radius if the pairlist update interval is determined statically at simulation startup.
r_{ic}	Interaction cutoff if the pairlist update interval is determined dynamically at runtime.
r_s	The small skin which applies to r_{ic} to determine the pairlist update interval.
r_{cl}	Long-range cutoff, $r_{cl} > r_c$, only in combination with the twin-range method.
N_s	The number of slices per region.
N	Usually the total number of particles (mostly atoms) if there is no special connection with GROMOS.
nragt	Total number of charge groups in the molecular system.
N_c	The average number of particles in a cell.
n_{s0}	Initial number of slices to communicate if domain decomposition applies.
$ncpc_{x/y/z}$	The number of cells per cutoff. Within GROMOS, ncpc is equal along all axis.
tnbx	The total number of cells.

Table 4.1 Survey of decomposition parameters

CHAPTER 5

Specification Model and Code Generation

This chapter introduces a domain-specific model which is applicable to replicated data parallel algorithms. Main ranges of application of that specification model are iterative algorithms intended to be partially parallelised. That part of the algorithm which is subject to parallelisation is specified as a control/data flow graph. A simple architecture model and the binding of algorithmic functions to hardware resources complete the specification. A code generator reads in such a specification thereby being able to generate most of the code containing interprocessor communications automatically, e.g. the parallel scheduler. The specification model and the code generator which are presented here follow the concept introduced in section 2.4.4.

5.1 Motivation

The parallel programming environment, introduced in section 2.4.3, was developed according to the GROMOS96P design goals. To be consistent, this concept was applied to implement the first hand-coded GROMOS96P prototype. This preliminary parallel version employed only synchronous (blocking) point-to-point communication using the *mpich*[1] message passing library. In accordance with chapter 4, the MD algorithm was decomposed spatially and the fully replicated data method was used. The parallelised part of the algorithm is embedded in the existing code as illustrated in fig. 5.1: the white parts of the flow diagram (left) represent the initial sequential implementation, the grey parts indicate where the new parallel code has its entry points. In addition to the parallel scheduler, an initialization function outside the main loop of the iterative algorithm is required, appearing as a grey box in the flow diagram labelled as "static data initialization". Essentially, this function distributes static data to all allocated computational nodes. This must be done once per program run and outside the main loop, because in contrast to dynamic data, static data remain constant throughout all iterations.

Many design goals could be achieved with this first GROMOS96P prototype. 1) The speed-up was in the range as predicted in section 3.4.2 for the *Net* architecture without broadcast. 2) Working with parameterized functions has proven to be efficient, especially the interplay of decompo-

1. mpich is a freely available, portable implementation of MPI [21], the Standard for message passing libraries. mpich is available for a wide range of platforms from Argonne National Laboratory, which is operated by the University of Chicago. http://www.anl.gov/

sition functions and parallel functions. 3) Parallel and sequential GROMOS implementations are derived from the same source code and 4) only two well defined entry points are required in the existing code. Nevertheless, crucial weaknesses must be pointed out:

- The parallel scheduler (fig. 5.1) is a highly complex function, including the superposition of two different control flows, the data flow of the algorithm, function calls, and all interprocessor communications. One control flow is that of the algorithm, the other steers the execution in dependence on the target architecture and the function mapping.
- Many dependences occur in the parallel code because the parallel scheduler and the parallel initialization function depend on the interface specification of parameterized functions. These dependences are unavoidable and intentional; in return, the handling with parameterized functions is significantly simplified, thereby granting the overall ease of use.
- The interface between parallel C code and the enclosing (FORTRAN) program is one of the main source of errors, since compilers are not able check the consistency of this interface parameter list.

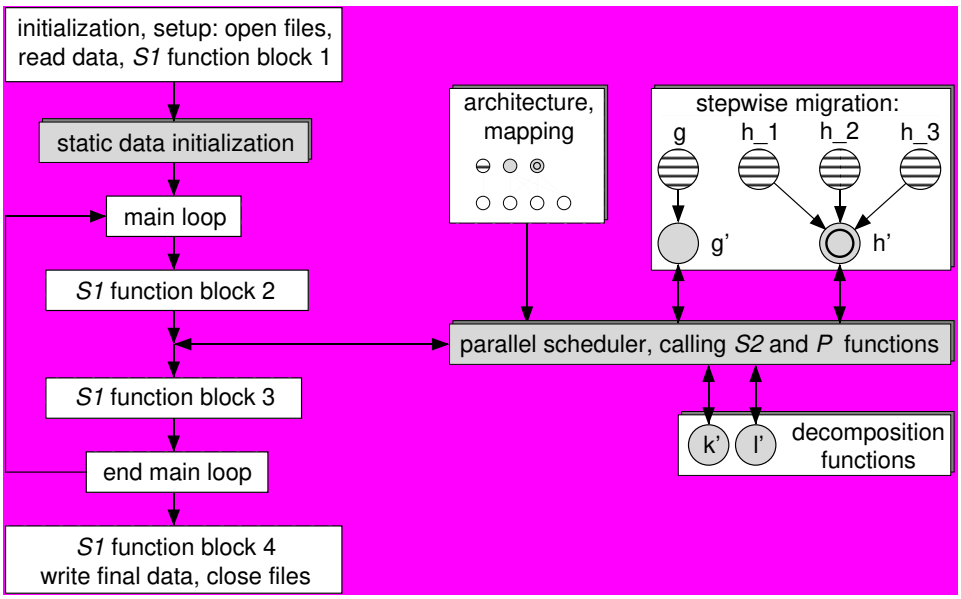


Figure 5.1 Proposed entry points of the parallel code. The left side depicts the block diagram of an existing sequential implementation as white boxes. Two parallel functions (grey) are inserted: an initialization function outside the main loop, and a parallel scheduler inside the loop which is invoked from the sequential scheduler. The parallel scheduler invokes parameterized functions (*S2* and *P* functions) in dependence on the control flow, the architecture and mapping function.

Whenever a parameterized function changes its interface, or a new function is added, dependences in the code become evident, making the prototype very susceptible to programming errors and difficult to change. To overcome these problems, and for the reasons given in section 2.4.4, a dedicated specification model and a corresponding code generator is developed. With this envi-

ronment it is possible to generate the most critical parts of the source code automatically, namely the parallel scheduler and the parallel initialization function, both of which depicted grey in fig. 5.1.

The specification model which is presented next allows to specify the parallel scheduler as a control/data flow graph (CDFG) independently of a target architecture. The remaining manual work includes the tasks of parameterization (stepwise migration) of sequential functions, developing decomposition functions, and the specification of the CDFG itself.

The development environment consists of three parts: *specification*, *verification* and *code generation*. The CDFG (fig. 5.3) combined with the hardware mapping (fig. 5.2) is called *specification* (section 5.2 and section 5.3). The *verification* step ensures the correctness of the specification and the availability of the resources, e.g. a communication library (section 5.4.1–section 5.4.3). The *code generator* finally generates C code automatically depending on the specification and the mapping on a specific target (section 5.4.4).

5.2 Parallel Software Concept

The domain-specific model initially has been developed to specify the parallelised part of the GROMOS96 MD algorithm. The model may naturally be used to specify any data parallel algorithms; the corresponding code generator is usable only if the targeted parallel system is running MPI. As a precondition, a replicated data decomposition scheme must apply.

5.2.1 Architecture and Function Mapping

An architecture is a set of computational nodes or processing elements (PE) interconnected with each other. Two types of PEs are differentiated: the *host* computer (H) and *client* machines (C*x*). An architecture always consists of one host machine and an arbitrary number of clients. As a minimum requirement, the host must be interconnected with each of its clients, whereas the underlying network is arbitrary. Client nodes never communicate directly with each other and are assumed to be identical. Violating this assumption, which is possibly the case if a heterogeneous workstation cluster is used, may lead to load balancing problems unless this heterogeneity is considered in the decomposition functions. The number of client nodes to be the only parameter in an architecture specification leads to a very simple structure superfluous to be specified explicitly. This very simple architecture model is used for the specification model and is not compliant to the abstract architecture introduced in section 2.2.3.

The mapping of algorithmic functions to computational resources is considerably restricted with the goal to limit the number of degrees of freedom in view of the code generator, and to keep the complexity of the hand-coded parallel scheduler within reasonable bounds. The binding is not determined for each function independently, instead, *function types* representing groups of functions are associated with computational nodes as a whole (fig. 5.2). Four different types of functions are relevant for the specification model and have been introduced so far (section 2.4):

1. Possibly generated functions: static data initialization and the parallel scheduler.

2. *S1* sequential functions are exclusively executed on the host and are depicted as striped white circles. *S1* functions usually refer to existing sequential code and are not subject to any kind of constraints. In accordance with section 2.4.3, *S1* functions are invoked solely from the sequential scheduler and are not allowed as functional nodes in the algorithm specification, thus they cannot be called from the parallel scheduler.

3. *S2* sequential functions—illustrated as grey circles—are parameterized functions following the guidelines which are described in the next section. *S2 algorithmic functions* are derived from *S1* functions to be available in the specification graph (stepwise migration, fig. 5.1). *S2 decomposition functions* determine the distribution of the parallel workload, hence they are normally not derived from *S1* functions but introduced as new functions. *S2* functions are called from the parallel scheduler and execute the same operation on the global replicated data structure, that way providing identical output data on each PE. *S2* functions are either executed on host only, or on the same nodes as *P* functions.

4. *P* functions are parameterized functions, typically derived from *S1* functions. *P* functions exploit the data parallel characteristic of the domain decomposition. *P* functions perform the same operation on different sets of data, that way generating different result data on each PE. *P* functions are either executed only on the host, on the clients only, or on all PEs.

For each node onto which the *P* functions are mapped, a disjunctive set of data (*region*) is determined by the decomposition functions (fig. 5.1). The regions denote the sub-domains on which the parallel functions work. It is the programmer's responsibilty to ensure that the parallelisation strategy of *P* functions coincide well with the decomposition functions. The algorithm decomposition should be chosen such that parallel functions have negligible overhead in case they are used for a sequential program compilation.

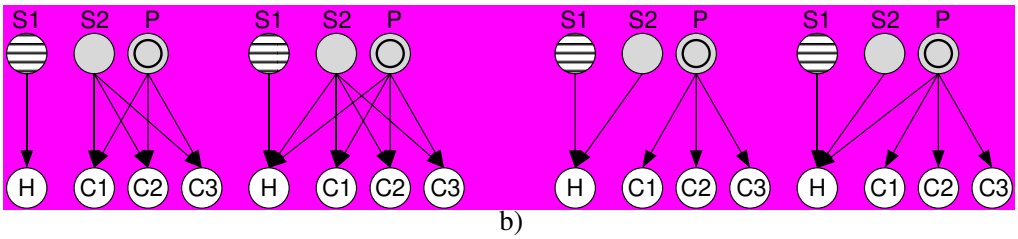


Figure 5.2 Four different examples show how three different function types (*S1*, *S2*, *P*) may be mapped onto hardware resources, with an exemplary architecture consisting of the host machine (H) and three clients (C1, C2, C3). *S1* mapping is always restricted to the host. The trivial case, where all functions are executed solely on the host, is not shown. The restricted mapping forces all functions which are declared as *P* functions to be mapped equally. The same rule applies to *S2* functions.
a) Two mapping possibilities for *S2* functions, if *P* functions are mapped to all clients except for the host.
b) Two mapping possibilities for *P* functions, if *S2* functions are exclusively mapped to the host.

The distinction of different function types is needed to classify different functional blocks within a specification. Communication and/or synchronisation points may be inserted only at functional block boundaries. A restricted mapping of functions onto PEs leads to efficient code

and simplifies the code generation process. The remaining mapping capabilities (illustrated in fig. 5.2) have proven to be sufficient for our applications. The number of client PEs is preserved as a parameter in the executable program, making a single program applicable to an arbitrary number of clients, including zero. Because adding or removing client machines does not change the code, the system is scalable a priori.

As a consequence of this mapping scheme two slightly different parallel schedulers are provided: a parallel scheduler running on the host (host scheduler), and a parallel scheduler running on all clients (client scheduler). In the course of the following discussion, the term *parallel schedulers* is still used if a certain topic concerns both host and client scheduler. Similarly, the static data initialization is performed by two different functions, one running solely on the host sending static data to all clients, and another running on the clients receiving the data.

5.2.2 Parameterized Functions

S2 and *P* functions are so-called parameterized functions and thus may appear in the specification model. Parameterized functions must meet the function interface standard and other constraints and requirements of the model:

- A standardised function prototype must be used ('i' and 'o' are reserved variables, the term *function* denotes the name of the function):

  int *function*(struct *function*instr *i, struct *function*outstr *o);

- The return value of type integer is zero if no error occurred, otherwise the return value signifies the corresponding error or warning number. Positive values refer to warnings, negative values denote errors.
- Parameterized functions may call standard C library functions. In addition, the files *function*.h and *specname*.h are always included. Any additional includes, for example message passing libraries, are not allowed. The term *specname* designates the name of the specification.
- The interface of a function is specified in *function*.h: the structures *function*instr and *function*outstr are declared in this file and contain all input and output data of *function*. Accessing external data from within *function* without using these structs is forbidden.
- *function*.h must provide a comment text where input struct members are declared either static (unchanged during program execution) or dynamic (explicitly communicated each time the function is called). A more comprehensive description of the function interface declaration is given in section 5.4.3.
- The include file *specname*.h contains function prototypes of all application specific functions. These might be timer functions for profiling purposes, FORTRAN standard library functions which do not exist in C language, and other application specific functions. Functions declared in this file are implemented in *specname*.c.
- The include file c.h is read in from *specname*.h. Primarily, c.h provides the mapping of parameterized functions to hardware resources. All global variables, if there are any, must be defined in c.h. In addition, a number of switches for various options are set here. The

most important is the switch determining if a parallel or sequential program is to be compiled. Less important options determine to what extent errors, warnings, and other information are reported.

- There are basically two possibilities to provide special non-parameterized functions which are accessible from parameterized functions but not from the parallel scheduler. The first possibility, as already mentioned, is to provide arbitrary functions in *specname*.h and *specname*.c. This option is advantageously chosen if the function in question is used by several parameterized functions. Secondly, it is of course possible to declare and implement functions within the source file *function*.c of a parameterized function. Because these additional functions must not be accessible to other parameterized functions, they have to be declared static.

Some further restrictions apply: pointers to functions and structs within structs are not allowed. Also, only simple data types[1] may be used in the function interface specification, and pointers to simple data types; pointers are always treated as arrays. All the rules mentioned above are mandatory if the code generator is used. An existing hand-coded version complies with the same rules.

5.2.3 Control/Data Flow Graph

From within the main loop, the parallel scheduler is called (fig. 5.1), which is specified with a CFG as in fig. 5.3a. The CDFG is a flat and directed control flow graph without loops, combined with a textual data flow specification. An algorithm specification consists of exactly one CDFG with dedicated single entry and exit points. The control flow graph differentiates three node types: functional nodes, conditional branch nodes and respective join nodes. The conditional nodes implement a standard if–endif statement. Edges between nodes represent execution dependences. The control flow execution semantics is the same as for the CDFG introduced in section 2.2.6. Other control structures like for loops and while statements are not supported. Task level concurrency is specified graphically, for example, *clcgeo* and *pfact* are task parallel functions (fig. 5.3a). In contrast to task level concurrency, data parallelism is implicitly specified by using *P* functions.

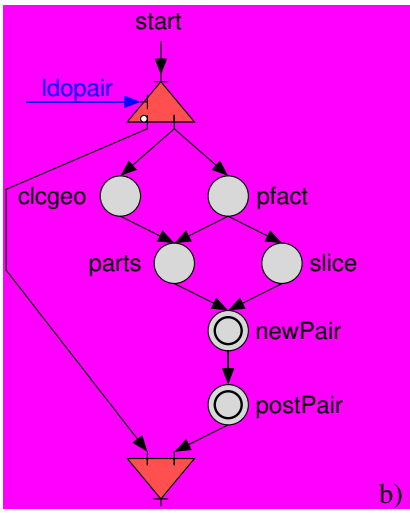
The function type (*S2*, *P*) is an attribute of functional nodes. Besides the entry and exit points of the graph, only *S2* and *P* functional nodes may occur. Entry and exit points represent the interface of the parallel scheduler. Analogously to algorithmic functions, dynamic data on these interfaces is accessible via the ‘i’ and ‘o’ variables. The interfaces of host and client schedulers will probably not be identical, because the host scheduler provides both control flow variables and data to be sent to functional nodes, whereas the client scheduler might only provide control flow variables on the interface. The same situation arises if the control flow reaches the exit point: if all result data is gathered on the host after each iteration, the ‘o’ struct of the client scheduler will contain no variable declarations.

The data flow and data assignments are not specified graphically but in textual form as in fig. 5.3b. Specifying the data dependences graphically would lead to very complex and possibly confusing graphs, because for each assignment one edge would be needed. Since there is not an obvious benefit from a data flow graph, data assignments are specified textually. Control flow and

1. Simple C data types: char, short, int, long, double, float.

data flow are strictly separated. Therefore, control flow variables like 'ldopair' may not be output variables of predecessor algorithmic *S2* or *P* functions but must be derived from input 'i' variables of the parallel scheduler. Input data for a particular functional node may be derived from any of the connected predecessor nodes. Within one iteration step, predecessor nodes are defined as those functional nodes from which there is a directed path of control to the current point of control. The input data assignment must be unique and data types must match. Data which is delivered to the specification via the 'i' variable of the host scheduler is accessible at any time on the host, client processors first have to communicate the data, unless the data item is also on the interface of the client scheduler.

a)



b)

```
...
newPair(1), Parallel, inputs {
      headl  = parts(1).headl: head;
      head   = parts(1).head;
      taill  = parts(1).taill: tail;
      tail   = parts(1).tail;
      xcgl   = clcgeo(1).xcgl: xcg;
      xcg    = clcgeo(1).xcg;
      boxl   = scheduleH(0).boxl: box;
      box    = scheduleH(0).box;
      nboxl  = pfact(1).nboxl: nbox;
      nbox   = pfact(1).nbox;
      pfact  = pfact(1).pfact;
      fslice = slice(1).fslice;
      nslice = slice(1).nslice
   } -> postPair(1);
...
```

Figure 5.3 Extract of the GROMOS96P specification, modelling the pairlist algorithm:
a) Control flow graph: a new pairlist is constructed if 'ldopair' is true.
b) Dynamic input data assignment of *newPair* (within braces) as part of the textual specification.

Referring to fig. 5.3b, dynamic input data of the function *newPair* is assigned to output data of predecessor nodes and input variables of the host scheduler (named scheduleH in fig. 5.3b). Each function has an instance number given in round brackets. The keyword 'Parallel' represents the functional attribute identifying a function to be of type *P*. An array is specified as pointer to a simple data type (this is not evident in the figure). Per array, two variables are required in the data flow: the length of the array as an integer (e.g. `headl`), and the array pointer itself (`head`).

5.2.4 Memory Management

This section describes memory management concepts which are used by generated and hand-coded parallel schedulers. First, the memory management strategy of the generated code is discussed. With the intention to improve the performance and to reduce the memory demand, the hand-coded implementation follows a slightly modified approach. The fundamental differences are shortly outlined.

Generated Code

Generated schedulers employ a very conservative memory management, in return, secure implementations without memory leaks are generated. Before a parameterized function is invoked, the respective arguments are either received or copied on all nodes onto which the function is mapped. The arguments are assigned to input variables of the function as soon as they are present and valid. After a (possible) function call, the memory which is occupied by the input arguments is deallocated in any case. Algorithmic functions and parallel schedulers have the duty to keep track of memory allocation and deallocation. A parallel scheduler allocates and deallocates the input variables of algorithmic functions. Analogously, algorithmic functions take control of their output variables. Output variables of algorithmic functions are initialized at program startup together with the static data initialization.

Outputs of a predecessor function are assigned to inputs of a function, with the exception of the scheduler input. The scheduler input serves as a data source similar to the output of an algorithmic function. Consequently, the scheduler output behaves like an input to a further algorithmic function. In case of a local array assignment, the array is always copied. First, the destination is checked if an allocation already exists. If this is the case, the memory is freed and newly allocated in order to provide the memory of the current copy operation. Simple data types are not explicitly copied since they are assigned directly. If a data assignment requires an MPI communication, the pointer to the receive buffer is assigned to the input variable of the target function, thus making additional copy operations superfluous. The receive buffers are always reallocated before new data is received.

Hand–Coded Versions

The hand-coded version also differentiates between assignments of single data types and arrays. If no communication is required, single data types are implicitly copied by direct assignment. In case of a local array assignment, the array pointer and length specifier are copied to the input variable, the array contents is not copied. If communication is required, a receive buffer must be allocated, and the same concept applies like for generated code: each time an array is received, a possibly existing former receive buffer is reallocated.

5.3 Language Specification

In order to provide a formal representation of an algorithm which has been specified in accordance with the concept of the previous section, a domain-specific language has been defined. The language includes the control flow graph (fig. 5.3a), the data flow specification (assignment statements, fig. 5.3b), and the hardware mapping (fig. 5.2). The textual representation of the control flow graph is written by hand or obtained from a CAD tool.

Within this section, only high level productions of the context-free language [65] are discussed and connected with the concepts of section 5.2; appendix C includes the complete syntax specification in the form of an EBNF. The top level production is the specification itself:

```
specList = "specification" "{" header funcList ifaceDefList declarationList
                               graphDef outMappingDecl "}"
```

A specification `specList` must start with the terminal symbol `specification`, followed by six subsequent parts enclosed in braces.

Specification header `header`

Within the specification header, the names of all four generated files are declared. These are: the parallel scheduler running on the host (host scheduler), the parallel scheduler running on all clients (client scheduler), and the two include files of the static data initialization functions. The file names must not have file suffix.

List of all Algorithmic Functions `funcList`

The second part of the specification includes a list of all algorithmic functions. These are, strictly speaking, the include file names including the search path. Again, the function names correspond to file names and are given without suffix.

Interface Declaration of both Parallel Schedulers `ifaceDefList`

Data on these interfaces comprise dynamic control flow variables and the dynamic I/O data flow. The interfaces are specified separately for the host and the client scheduler. Control flow variables must be provided on both interfaces. Dynamic input data may be omitted on the client scheduler interface, since MPI communications are inserted automatically to distribute the data.

A list of variable declarations is composed to a C structure very similar to the declaration of an algorithmic function interface. If a specific variable represents a dynamic array, the variable identifier is prefixed with an asterisk. Each variable declaration of the list is followed by the additional information determining if the variable is of type static or dynamic. A static variable is not modified within the main loop (fig. 5.1) and therefore never communicated. In contrast, dynamic variables *do* change every iteration and are communicated if required. This semantics of a static variable declaration differs from that of the C language standard.

Global Parameters `decalarationList`

Two global parameters must be specified, represented by the terminal symbols `mapping` and `initSwitch`, respectively. The parameter `initSwitch` is of type integer and controls the initialization of the input data of algorithmic functions. If set to non-zero (default), initialization is performed each time the parallel schedulers are invoked, viz single data types are set to zero, all memory is deallocated and the respective pointers are initialized. The enumeration parameter `mapping` determines the mapping of *S2* and *P* functions, in accordance with fig. 5.2; the enumerator can take four different values:

- `AllonC`: *S2* and *P* on all clients, nothing on host, fig. 5.2a, left.
- `AllonA`: *S2* and *P* on all computational nodes, fig. 5.2a, right.
- `AllbutSonC`: *S* solely on host, *P* on all clients, fig. 5.2b, left.
- `AllbutSonA`: S solely on host, P on all computational nodes, fig. 5.2b, right.

Graph Definition `graphDef`

The graph itself is specified node per node in an arbitrary sequence. The language differentiates three node types, namely `condition`, `endcondition`, and `function`.

A conditional node `condition` requires two boolean expressions; the first is evaluated on the host, the latter on the clients. Although user defined constants are allowed, in the general case control flow variables are associated with those boolean expressions. Because control flow variables must be derived from the interface declaration `ifaceDefList`, they do not have an

instance number. Two lists of successor nodes follow the branch conditions. The first list is processed if the boolean expression is true, the latter otherwise. Each conditional node must have a unique instance number and a matching `endcondition` with the same instance number.

Each functional node `function` represents an algorithmic *S2* or *P* function call, the list of input parameters for that function (`mapAssignmentList`), and a list of successor nodes. First, the name of the function is given, followed by the function type (`Sequential` for *S2* functions, `Parallel` for *P* functions). The production `mapAssignmentList` specifies the data sources which are associated with specific input variables. One data assignment is composed as follows (refer to the EBNF given in appendix C): the identifier of the input variable is optionally followed by a second identifier, separated with a colon. The second identifier refers to a special *collect* function. The collect function specifies how data is gathered if the current function is mapped solely to the host, and the function providing the source data is executed on the clients or on all nodes. After the equals sign, the data source is given, followed by an optional list `lengthList`, separated with a colon. The `lengthList` is either empty or contains a list of input array names, for which the current variable determines the length.

Four types of data sources are possible, namely the outputs of another algorithmic function, a constant number, or one of the internal reserved variables `mpiId` and `mpiNumNodes`. If the source is an output of a predecessor node, the instance number of the source node is required. This is a regular data assignment according to the data flow of the specified algorithm. The user is responsible assigning only valid data. If, for example, a certain branch of the control flow has not been executed, associated data might be outdated. In this case, the implementation of the algorithmic function must ensure that the outputs remain valid. The variables `mpiId` and `mpiNumNodes` both are MPI implementation specific and represent the local node number and the total number of allocated computational nodes, respectively.

Definition of the Output Mapping for both Schedulers `outMappingDecl`

The last part of the specification determines the data which is fed to the output interface of the parallel schedulers. Two identifiers are expected which must coincide with the names of the schedulers (defined in `header`), first the host scheduler, secondly the client scheduler. For each scheduler the output data fields, as defined in the third part of the specification (`ifaceDefList`), are associated with outputs of algorithmic functions using a `mapAssignmentList`. Mainly here collect functions will be necessary to ensure that the results are transferred correctly to the host. The output mapping may be empty.

5.3.1 GROMOS96P Specification

The complete control flow graph of the GROMOS96P parallel algorithm and the corresponding textual specification are to be found in appendix D.

5.4 Automated Processing

5.4.1 Specification Parser

The scanner reads the specification and generates a sequence of tokens. If a symbol is recognised, the respective token—an internal data type—is generated and added to the output sequence. For example, if a number has been recognised, the token NUMBER is generated. If a character string is not a known symbol and does not contain illegal characters, it is treated as an identifier. Some tokens may carry attributes. For example, the attribute of the NUMBER token is the value, and the token IDENT (identifier) has the recognised string as an attribute.

The token sequence is fed to the specification parser. The recursive descent parser checks the syntax of the token sequence in accordance with the language specification. If the parser starts up with a token sequence, a correct specification is assumed. For each token the parser tests if the current production is fulfilled. The main goal, the production `specList`, is suspended if a non terminal symbol is found. Instead of the suspended production, the new non terminal symbol is explored first. This leads to a recursive algorithm: starting from the top level goal, the parser steps down the production hierarchies.

The parser controls the whole data processing of a specification, thus calls scanner functions as well as semantics tests.

5.4.2 Checking the Static Semantics

Once the parser has identified a syntactically correct part of a specification, different methods and functions are invoked to ensure the consistency of the specification, to find conflicts, and to transform the input data into internal data structures. If one of the checks fails, the parser aborts execution and provides an appropriate message. This section enumerates the most important semantics analysis functions.

- If the `header` is successfully read, the function names of generated functions are checked if they are disjunctive and copied to internal data types. Possibly already existing functions are overwritten without issuing a warning.
- If the function list `funcList` is processed, for each *function* in the list it is checked if there is an include file *function*.h. Then, the C code parser is activated to process the interface declaration of the algorithmic function, which is specified in *function*.h (section 5.4.3).
- The parallel scheduler interface declaration list is checked if the names of the given schedulers are consistent with those defined in the header section. It is checked whether all interfaces are declared. The scheduler interfaces are included in the internal graph representation enabling them to be sources for input data assignments.
- Include files for the schedulers and static data initialization functions are generated, because at this point interface specifications of all functions are processed. These include files mainly consist of modified I/O interface declarations of the algorithmic functions. Modifications concern the type-qualifiers of input variables.

- Because the graph definition is read node per node in an arbitrary sequence, a structural consistency check may not be performed until the complete control flow graph is read. The tests which are executed on the complete control flow graph ensure the absence of forbidden control dependences. Moreover, it is tested if all specified functional nodes are part of the graph. Data dependences are not checked against the control flow. It is therefore possible to specify a successor node of the current function as source of an input assignment. It is only checked if the specified source data item really exists. In addition, minimal checks apply to length specifiers of arrays including only the verification of the correct data type. Because not yet implemented, it is not tested whether the referenced data item is indeed an array, and if that array is really assigned.
- It should be noticed that providing a collect function or a `lengthList` is *syntactically* optional. An array with missing length specifier is accepted without issuing a warning. If the collect function specifier is missing in a specification, the appropriate code to communicate data to the host is not generated.
- Analogous to data assignments of algorithmic functions, the processing of the output mapping of the schedulers (`funcList`) aborts if a non existing source data item is referenced.

5.4.3 Processing the Interface Declaration of Functional Nodes

Parameterized functions may not access global data structures, hence external data must be read or written via the *function* interface, which is specified in the *function*.h include file. The only exception are global constants which are defined in the file c.h. For example, the GROMOS specification defines an integer constant *Decomposition* denoting the axis along which the simulation box is decomposed.

The specification parser invokes a dedicated C code parser [59] to process the interface specification of a parameterized function. An interface specification consists of two parts:

- The declaration of two static structs: *function*instr and *function*outstr. The ‘static’ (storage class) struct declaration limits the scope of that object to the rest of the source file being compiled. All struct members of *function*instr must be declared ‘const’ (type-qualifier), because the input data of *function* may not be altered within the function body.
- A *special comment text* is identified if the first non-space type character of the comment is a dollar sign. The special comment in each *function*.h file includes the information which is required to distinguish static and dynamic input variables of *function*instr [59].

The file *function*.h is exclusively included by the file containing the implementation of *function*. However, the interface declaration of all parameterized functions must also be available to the parallel scheduler in order to assign dynamic variables. Simply including all *function*.h files in the parallel scheduler would not help, because respective input variables are declared ‘const’ thus are not modifiable.

The specification parser collects the information of all functional interfaces and *generates* specific include files for the parallel scheduler and static data initialization, containing the I/O struct declarations of all parameterized functions. Declarations of output structures are simply copied, whereas input struct members are partially modified: in case the include file of the parallel sched-

uler is generated, ‘const’ type-qualifiers are removed for all dynamic input variables. The same procedure applies for static input variables if the include file of the static data initialization function is generated.

5.4.4 Code Generation

Once a specification has been successfully parsed, an internal data structure is available including the control flow graph, the data assignments of the data flow, and the interfaces of all the involved parameterized functions. Since the include files have already been built while parsing, the remaining task of code generation concerns the parallel schedulers. The employed memory management concept (section 5.2.4) simplifies the code generation process significantly, in return, the memory demand of generated code inflates, and copying large arrays may increase the overall computation time of an implementation.

The code generator considers the mapping of parameterized functions to computational nodes as well as the function type (*S2*, *P*) in order to determine the computational node(s) onto which a function is executed. For each data assignment the code generator first has to determine if the assignment is local or remote. If it is remote, MPI communications are inserted; if it is local, a copy operation is inserted. In order to ensure that successive MPI-sends to the same destination are received in the correct order, the communications are numbered continuously by using MPI tags [21]. In the opposite case data is gathered on the host, thus a *collect* function determines how the global data structure is to be restored. This function is called as many times as there are clients in the system, and solely from the host scheduler. The collect function executes a user specified binary function and provides an array on its output which is copied to the parameterized destination function. A binary function accepts two arguments as input, performs the defined associative operation and provides the result on the output. The function prototype of the *collect* function is chosen compliant to the MPI standard, hence a number of predefined MPI reduce functions may be used.

Generated code is compiled with a standard C compiler, the application is linked using a standard C or FORTRAN linker. This procedure has been described in section 2.4.3.

5.5 Summary

In this chapter, a specification model for data parallel algorithms and an appropriate C code generator have been developed. The presented programming environment and the respective tools are especially best suited for iterative algorithms but may be applied to any replicated data parallel algorithm with the goal to simplify parallel programming and to increase its reliability. In order to employ the new model, the parallel part of the algorithm must be separated from the master application and specified with the presented model.

This novel development environment for data parallel algorithms comprises the following achievements:

- The introduced domain-specific specification model allows the user to formally describe data parallel algorithms as a control/data flow graph. A language has been developed in order to represent such a graphical specification in textual form.

- A toolset consisting of a specification parser, a dedicated C code parser and the code generator does process the algorithm specification and the I/O interfaces of all functions referenced in the control flow graph.
- The code generator is able to generate a parallel scheduler taking into account the control flow of the algorithm and the parallel target architecture. All needed interprocessor communication is generated automatically.
- Before the main loop of an iterative algorithm is entered, the static data of parallel functions is distributed to the clients. In order to support the coding of the initialization functions, include files containing the declarations of all static data are generated.

The presented specification model and its realisation within the parallel development environment is naturally restricted to the algorithm class for which it has been designed. Therefore, the data parallel algorithm *must* employ the replicated data decomposition method. In order to reduce the complexity of the code generator implementation, its functionality has been limited to the essentials. Firstly, task level concurrency is ignored: the code generator builds the execution sequence of two parallel tasks in the same order as they are read by the specification parser. Secondly, the generated code employs an inefficient but secure memory management thus requiring much temporary memory. Finally, the environment does not guarantee a correct specification although the semantics of the control/data flow graph is well defined.

The major consequences for the MD algorithm designer may be summarized as follows:

- The user is working with an abstract model of the algorithm, thus the presentation of the algorithm is clearly apparent and not dependent on the target platform. Hardware dependences are strictly separated from the algorithm specification. The toolset also handles the communications between algorithmic functions and processors.
- Algorithmic functions as part of a specification are used in sequential and parallel programs. The algorithm designer provides only these algorithmic functions and is relieved from keeping track of data and code dependences.
- Due to the lack of semantic tests a faulty specification is not necessarily recognised when the code is generated. This problem is alleviated because in most cases the code of faulty specifications provokes compile errors.
- The interface between generated parallel C code and the embracing FORTRAN program continues to be one of the main error sources.

CHAPTER 6

Performance Comparison

The performance of three different parallel GROMOS96P versions is evaluated and compared with GROMOS96 profiling results of section 3.2. First, the differences of the three versions are pointed out, followed by the definition of the performance metrics. Section 6.3 comprises a detailed compilation of the measurements. The investigated parallel platforms are shortly described in section 6.3.1.

6.1 GROMOS Implementations

The first GROMOS96P prototype was implemented with simple synchronous communication using the *mpich* message passing library based on the MPI standard (Message Passing Interface, [21]). After some performance bottlenecks have been pointed out, the communication scheme was changed. The current hand-coded version now obeys an *asynchronous communication scheme with optional barrier synchronisation*.

6.1.1 Hand–Coded Asynchronous MPI Implementation

The current implementation distributes the data with non-blocking point-to-point communications. MPI reduce functions are used for collective communications at the end of a time step. For example, a collect communication is required if the output of a *P* function (assumed to be executed on all nodes) is assigned to the input of a *S2* function (assumed to be executed on host only). In case of the GROMOS96P algorithm, only one *S2* function may cause such a communication, namely *prpxrSolu* (refer to the GROMOS algorithm specification in the appendix, fig. D.1).

Barrier Synchronisation

Due to the fact that *S2* and *P* functions are not necessarily mapped equally (fig. 5.2), communication and/or synchronisation may be required at certain points within the control flow specification. These points are referred to as barrier synchronisation points and are defined by functional blocks. Functional nodes interconnected *directly* by the control flow may be grouped together to build a functional block if 1) all functions are of the same type, e.g. *S2* 2) all functions in the block have the same set of possible predecessor nodes outside the block. In this context, a *direct* control flow interconnection between two functional nodes may cross a conditional branch or join

node. Interprocessor communication possibly occurs at points where the control flow penetrates functional block boundaries. Similarly, optional barrier synchronisation points are inserted if a control flow edge enters a functional block.

Whether the barrier synchronisation is to be enabled or not must be decided at compile time. Regardless of the setting of the barrier-switch (appendix B.2), but in dependence on the data flow, implicit synchronisation may occur at barrier synchronisation points: if a data communication is of type *collect*, data is possibly sent from the clients to the host, synchronising all nodes even with a disabled barrier.

The input of the GROMOS function *prpxrSolu* is derived exclusively from host scheduler thus implicit synchronisation never occurs. Consequently, if the barrier synchronisation option is disabled, all schedulers are running completely asynchronous throughout the parallelised part of the algorithm. In order to measure the runtime differences of the clients with respect to the host, a synchronisation of all nodes is enforced at the end of the parallel scheduler, immediately before the collective communication.

The hand-coded programs are referenced to as *promd.pa* and *promd.ps* representing the version without and with barrier synchronisation, respectively.

6.1.2 Generated Synchronous MPI Implementation

The generated version differs from the hand-coded version in that only blocking synchronous point-to-point communication is used. Additional barrier synchronisations may not be specified. Moreover, the generated version employs a different memory management of dynamic data flow variables (section 5.2.4). The generated version is referenced to as *promd.pg*.

6.1.3 Sequential Versions

Two types of sequential versions are separated: 1) the original GROMOS96 programs *promd.32* and *promd.64* representing the single and double precision version and 2) the GROMOS96P version, which is easily derived from the hand-coded source code of the parallel version since special keywords have been introduced to identify parallel codes. The sequential GROMOS96P program makes use of exactly the same algorithmic functions as the parallel does. In section 6.3.2 it is shown that only a moderate performance gain is to be expected of such a dedicated sequential GROMOS96P version.

6.2 Performance Metrics

As a precondition, the same function mapping is used throughout: all *S2* and *P* functions are executed on all computational nodes.

In order to measure the execution time of parallel functions timer functions providing similar features like a stop watch (start-stop, reset) have been used. The timers are based on operating system- and MPI-time functions providing execution times in different resolutions: the MPI-timer provides the time as a floating point number in units of seconds; the system-timer delivers an integer value, either in units of seconds or in units of ten microseconds.

Timer functions are exclusively used by the host machine thus determining execution times of functions running on the clients is not possible. To profile a certain algorithmic function which is called from the parallel host scheduler, a timer is started before the function is called and stopped immediately after the function returns. The total time which is spent within the main loop of the algorithm is determined by measuring that time on the host. In case of GROMOS, this is done in subroutine *runmd.f* (section 3.2.2).

Table 6.1 summarizes the measured values and gives a short description. Function names refer to algorithmic functions of the GROMOS specification graph (fig. D.1).

Function	Explanation
solute	Accumulated execution time of *soluForce* on the host.
solvent	Accumulated execution time of *solvSolv* on the host.
pairlist	Accumulated execution time of the pairlist calculation on the host, including the functions *clcgeo*, *pfact*, *parts*, *slice*, *newPair*, *postPair*.
integration	Accumulated execution time of the integrator (*mdleap*, table 3.2).
sync	All clients are synchronised before the host initiates the collective communications. The *sync* timer is started as soon the host reaches the synchronisation point and is stopped if the slowest client has reached the synchronisation point. This synchronisation point is mandatory (section 6.1.1) and additional to barrier synchronisation. This time must be (almost) zero for the program *promd.pg* since generated code is based on synchronous communication.
main	The total time spent within the main loop.
rest	This time is the difference between *main* and the sum of all accumulated functional execution times described above and *sync*: $rest = main - (solute + solvent + pairlist + integration + sync)$ (6.1) Consequently, *rest* comprises of the execution times of 1) all other algorithmic functions within the main loop, including bonded forces, the Virial preparation, etc. 2) all communications including barrier synchronisation (if enabled) and idle times possibly induced by load imbalances.
deviation	*main* is measured with two different timer functions both providing a time in units of seconds (MPI timer and operating system timer). The *deviation*-time is the difference between these two *main*-times and may serve as a rough criterion of assessment of the reliability of measured times.

Table 6.1 Performance metrics: accumulated execution times of different algorithmic functions, the total time spent within the main loop (*main)*, the accumulated time lost due to synchronisation (*sync)*, and the derived quantities *rest* and *deviation*.

6.3 Results

6.3.1 Test Environment

Two performance benchmarks which are referenced to as *thr2* and *thrG2* are used. The benchmarks each represent a certain molecular system and a simulation parameter set specified in section 3.2.1 and appendix A.

In the course of the following discussion, three parallel computing systems are investigated to determine the delivered performance (particular machines are described in appendix B):

1. A Sun Solaris workstation cluster consisting of 1-5 SunUltra10 is interconnected with FastEthernet. The *mpich* message passing library is used.
2. The heterogeneous workstation cluster consists of a SGI-Octane workstation as host machine connecting 1-4 Alpha-workstations running DecUnix. The clients are connected with the host by a switched Myrinet network. The *mpich* message passing library is used with a standard device driver using protocol stacks.
3. IBM SP-2, 1-44 nodes.

6.3.2 Comparison of the Execution Time

Table 6.2 provides a survey of execution times of all GROMOS versions running on one computational node.

GROMOS	SunUltra10	SunUltra30	SGI Octane	Alpha 21164	SP-2
96: *promd.32*	1815	2653	2193	2106	3044
96: *promd.64*	2035	3109	2193	2106	3632
96P: *promd.pa*	1595	2384	2145	2008	3047
96P: *promd.ps*	1771	2543	2156	2006	2971
96P: *promd.pg*	1660	2569	2165	1931	3475

Table 6.2 Single processor execution times of 100 *thrG2* steps on different platforms, times are given in units of seconds.

In the following, the new parallel GROMOS versions are referenced only with their suffix, e.g. *pa44_14* denotes a simulation with the program *promd.pa*, executed on 44 computational nodes applying a 1.4nm cutoff. The results of table 6.2 are shortly discussed:

- Despite the parallelisation overhead GROMOS96P versions are faster than GROMOS96. This is due to the new pairlist algorithm replacing the brute force approach of GROMOS96.
- The execution time is not necessarily increased by performing double precision floating point arithmetic (SGI Octane, Alpha 21164). This fact represents the general trend in the development of new general purpose microprocessors and should also be considered if computational software is developed.

- A dedicated sequential version of GROMOS96P could easily be derived from the hand-coded versions since special keywords have been introduced to identify parallel codes. This is not necessary at all because firstly all GROMOS96P versions outperform the sequential GROMOS96 and secondly a dedicated sequential GROMOS96P version has shown only a small gain of about 5%.

6.3.3 Exploring the Scalability on the SP-2

The following performance tests were done in order to asses the quality of GROMOS96P: 1) the delivered performance of the hand-coded version is determined in dependence on the barrier synchronisation option 2) the scalability of parallelised functions is investigated and 3) speed-ups are calculated with respect to a real world Molecular Dynamics simulation.

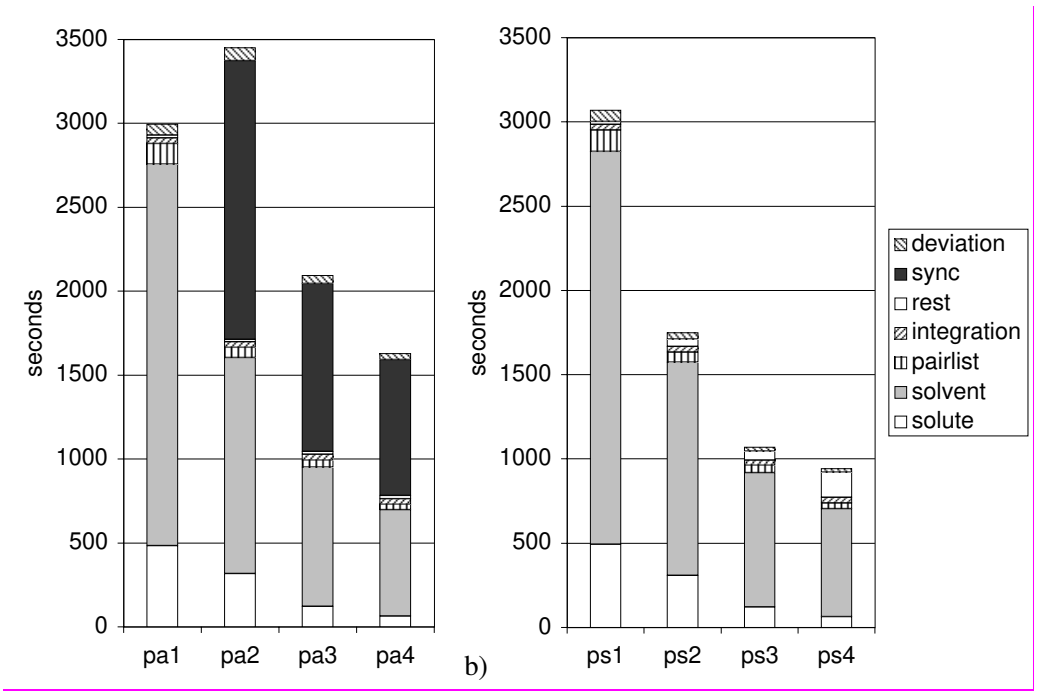


Figure 6.1 GROMOS96P hand-coded version running the *thrG2* benchmark on the SP-2 with 1-4 computational nodes.
a) Asynchronous data communication without barrier synchronisation (section 6.1.1).
b) Asynchronous data communication with barrier synchronisation.

Profiling results of the programs *promd.pa* and *promd.ps* are presented in fig. 6.1 for 1-4 processors. This comparison—principally concerning the barrier synchronisation technique—allows to conclude the following:

- The scalability of parallelised functions (*solute*, *solvent*, *pairlist*) is nearly ideal for a small number of processors (fig. 6.1).
- Without barrier synchronisation (fig. 6.1a), the host spends approximately the same amount of time to wait for the clients reaching the (enforced) synchronisation point as the host has been taken to perform the local calculations (6.2):

$$sync \approx solute + solvent + pairlist \tag{6.2}$$

To explain equation (6.2), the SP-2 communication subsystem obviously does not start an asynchronous data transmission until the host reaches its synchronisation point. This inexplicable behaviour reduces the reachable speed-up of *promd.pa* by a factor of two.

- The share of communication time to the overall execution time increases as the number of processors increases. This is visible in fig. 6.1b only, since communication times are hidden if barrier synchronisation is disabled.
- Consequently, the large scale studies on the SP-2 (fig. 6.2 and fig. 6.3) are carried out with enabled barrier synchronisation.

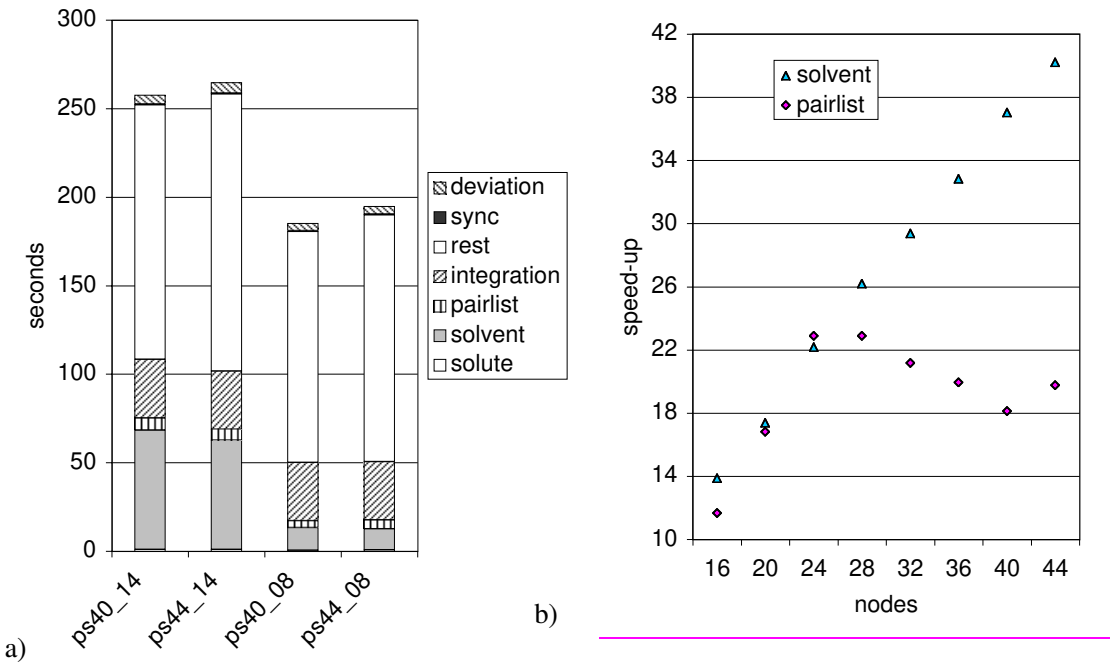


Figure 6.2 The GROMOS96P hand-coded version running *thrG2* on the SP-2 with a large number of processors identifies the performance-limiting factors: the share of communications and *integration*.
a) Communication overheads on the SP-2: the performance loss due to communication is independent of the cutoff radius.
b) Scalability of parallel functions: solvent-solvent interaction calculation and pairlist construction (rectangular box, cutoff 1.4nm). The speed-up is calculated as the execution time of the respective function on one node divided by the time to process one *region* (section 4.3) on the host.

- A short cutoff decreases the number of interaction calculations per processor, whereas the communication due remains the same because of the replicated data decomposition. This is illustrated in fig. 6.2a showing results with 0.8nm and 1.4nm cutoffs, respectively.

- The scalability of parallel functions is almost ideal up to 40 nodes and more (fig. 6.2b, *solvent*), except for the pairlist algorithm. In consistence with previous findings, the pairlist algorithm is most efficient if the number of grid cells approximates the number of particles in the system (section 4.1.3). Considering the *thrG2* benchmark executed on 44 nodes (fig. 6.2b), more than 85,000 grid cells are opposed to 12,246 particles, resulting in many empty cells and thus a bad efficiency of the pairlist algorithm.
- Fig. 6.2b and fig. 6.3 show that the scalability is not limited by the algorithmic overhead of parallelised functions but by the execution time of the sequential part of the program and communication (3.1). The remaining sequential part of the program consists not only of *integration* but also of some codes within the main loop (bonded forces, SHAKE, etc.) which contribute to *rest*. The amount due to these codes depends on simulation-specific options and is approximately half of the *integration* execution time in the present case.
- By using the performance metrics introduced in section 6.2, speed-up calculations for *solute* would be distorted due to density fluctuations of solute charge groups. Therefore, the speed-ups for *solute* are not provided in fig. 6.2b.
- Communication *is* the limiting factor if the number of processors increases. This is consistent with the assumptions made in the performance estimation section 3.4.2 namely disregarding the algorithmic parallelisation overheads.

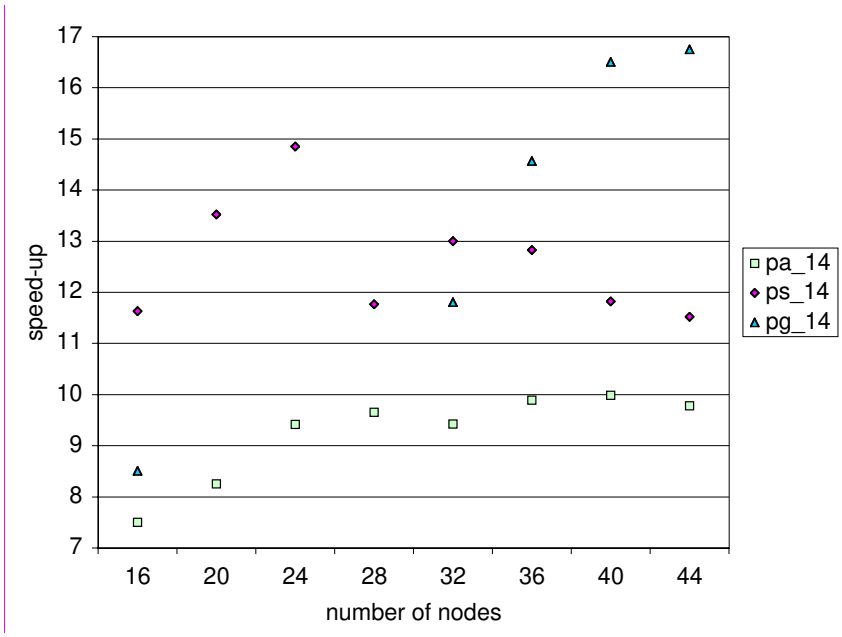


Figure 6.3 GROMOS96P speed-ups: *thrG2* benchmark on the SP-2.

- The speed-ups reached by different parallel GROMOS versions (fig. 6.3) do approximate the predictions carried out in section 3.4.2, refer also to fig. 3.7a. Although the behaviour of the SP-2 communication subsystem is somehow unpredictable if asynchronous communication is used (*pa_14* and *ps_14* in fig. 6.3), respective implementations always outperform the synchronous alternative. More thorough investigations of this communication issue must be a subject of ongoing work.

6.3.4 Workstation Cluster

The performance and communication behaviour of two different workstation clusters with five machines each are compared.

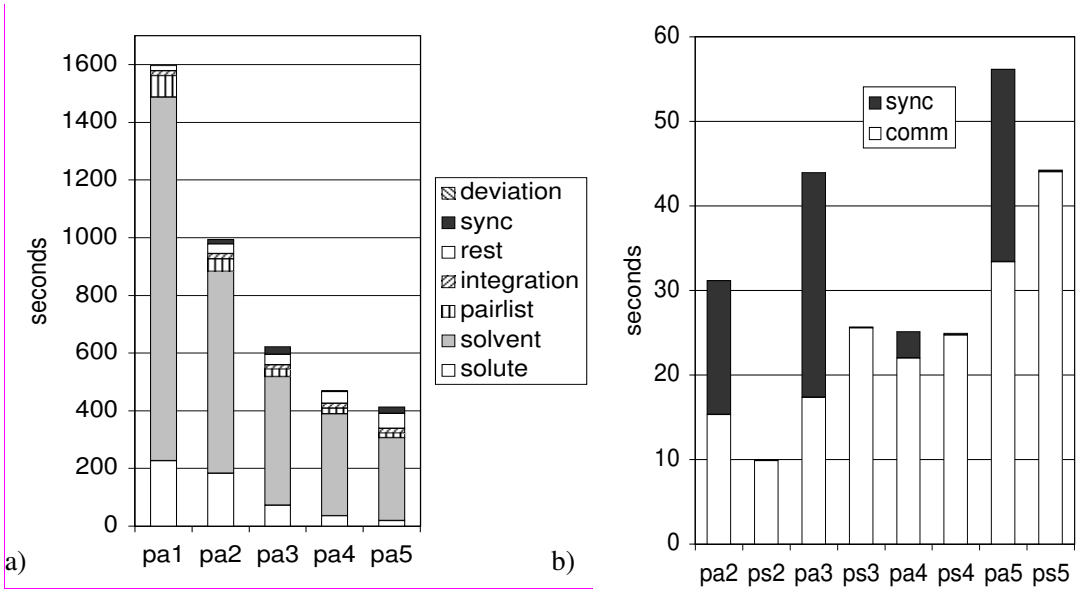


Figure 6.4 *thrG2* benchmark running on the SunUltra10 workstation cluster (FastEthernet).
a) Execution time of single functions and communication dues without barrier synchronisation.
b) Comparison of communication times in dependence on the barrier synchronisation. The time *comm* is that part of *rest* representing the communication share only.

- The SunUltra10 cluster has been tested in an office environment. Fig. 6.4b visualises the influence of the synchronisation method which is obviously of minor importance (compared with the SP-2). A better performance is not achieved by omitting synchronisation.
- The SGI/Alpha cluster communicates over a dedicated Myrinet network. The influence of the synchronisation method is negligible. Experimental results allow the conclusion that the communication subsystem is able to perform blocking communication only.

	Parallel GROMOS	**Speed-up in Dependence on the Number of Clients**			
		1	**2**	**3**	**4**
SGI/Alpha	*promd.pa*	1.8	2.7	3.5	4.1
	promd.ps	1.8	2.7	3.4	4.1
SunUltra10	*promd.pa*	1.6	2.6	3.5	3.9
	promd.ps	1.9	2.8	3.6	4.2

Table 6.3 GRMOS96P Speed-ups on different workstation clusters, running the *thrG2* benchmark. Considering small workstation clusters, the achieved overall performance is the same for both networks investigated: a dedicated Myrinet and FastEthernet in an office environment.

CHAPTER 7

Conclusions

7.1 Fundamental Results

The goal of the present thesis was to compare and improve different design space exploration methods to find the optimal parallel architecture, a programming model for a given Molecular Dynamics (MD) algorithm, and to verify the quality of these methods by one concrete implementation. Essential results are:

- The usefulness of *system synthesis* using an Evolutionary Algorithm (EA) to find suitable implementations for a given parallel algorithm strongly depends on the decomposition scheme of that algorithm. It has shown that EA-based design space exploration is not advisable for a data parallel algorithm due to its lack of sufficient task level concurrency.
- Existing parallel computation models like the PRAM or logP model have shown serious weaknesses in modelling parallel architectures on a higher level of abstraction. Thus the logP model has been adapted in order to cope with the communication scheme of multiple processors based on *message passing*.
- Analytical *performance estimation* of possible target architectures is based on the parallel computation model and generic functional models of the considered algorithm. Performance estimations, as have been executed within the frame of this thesis, allow to quickly assess design alternatives, each consisting of a certain parallel architecture combined with the algorithm model applying different MD parameter sets. As a result, *dedicated hardware* approaches have proven to be the first choice if only the delivered performance is to be considered.
- However, if the emphasis is on using a finished specialised hardware product for MD simulation studies, the design of such a parallel computer is difficult to justify. The most difficult constraint to meet in a hardware project is the time constraint. With the power of microprocessors roughly doubling every two years, a parallel design may well be out of date by the time its construction is completed and, unless some easy migration path is envisaged in the design, the software written specifically for the machine becomes useless. Consequently, the focus of research must be on the question how to provide *most efficient software solutions* independently of the underlying hardware, taking account of possible future hardware extensions.

- Standardised message passing libraries like MPI are generally used to handle interprocessor communication. MPI offers an abstracted communication interface which is used in the same manner on all supported architectures. Similar to parallel computing models, parallel software development also moves to higher levels of abstraction with one of the goals being the synthesis of parallel programs from behavioural specifications.

  Initially developed as a part of the MD programming environment, a domain-specific language may be used for any kind of replicated data parallel algorithms. The appropriate code generator is capable to handle all interprocessor communications and has been successfully tested for the MD algorithm. In addition to that, the user is relieved from keeping track of data and code dependences in parallel implementations.

- The search for an adequate decomposition strategy for the GROMOS MD simulation package has been proven to be rather difficult. A profound study has been carried out in order to come to a design decision, pointing out the replicated data method to be the only solution compliant to all GROMOS features. The main advantages of replicated data decomposition are: 1) the possibility of automated processing and 2) the most simple parallel programming. In return, however, the scalability will be limited due to communication overheads. The currently available parallel GROMOS prototype scales well up to about 20 computational nodes and reaches its maximum performance with about 30-40 nodes.

7.2 Future Perspectives

Based on this work, promising areas for future work and research may be:

- Extending the parallel programming environment by an enhanced specification model coping with a different decomposition paradigm. A broadened code generator functionality would then enable automated processing of other than the replicated data decomposition scheme, thereby requiring additional communication capabilities made available to the code generator, for example point-to-point communications between all computational nodes for a domain decomposition.

- Experimental results on different parallel architectures have shown a varying behaviour of the communication subsystems if asynchronous message passing communication is used. For certain architectures this is expressed by extremely long communication times far off the optimum. Thus it might be interesting to

  1. investigate the network infrastructure of these architectures and
  2. enrich the specification language by additional attributes enabling the user to control the generation of communication codes. Additional attributes would allow to selectively choose between synchronous, asynchronous or broadcast communications.

- The parallel GROMOS implementation model may be refined: the spatial decomposition algorithm would be subject to optimisation by decoupling the pairlist construction from load balancing. In addition, the twin-range pairlist method should be incorporated in the parallel code.

Appendices

A Benchmark Description

Two types of benchmarks are separated: performance benchmarks and molecular systems for code validation. Performance benchmarks are used to assess the performance of the parallel software against the initial sequential GROMOS96 distribution. Validation examples have been carefully chosen with the goal that as much of the modified code as possible is executed in order to expose possible programming errors.

Performance benchmarks

Both performance benchmarks consist of the same solute molecule (Thrombin) and the same solvent (water). They only differ in that the simulation box of one benchmark is a truncated octahedron (*Thr2*),whereas the other is a rectangular box (*ThrG2*). As an option, the solute may be simulated without water (Thr1), that is in vacuum without periodic boundaries (table A.1). If not otherwise indicated, the simulations are run with the following settings:

- 100 time steps in three dimensions and with double precision floating point arithmetic.
- The SHAKE algorithm is used to handle constraints and the Virial is calculated.
- Final coordinates and energies are written to files.
- print block: 49 49 0
- write block: 49 0 0 49 0 1
- a pairlist is constructed every five steps, with a cutoff of 1.4nm, without twin-range.

With the configuration described in appendix B, the *ThrG2/promd.64* benchmark occupies 34MB of the main memory.

Molecular Systems for Code Validation

Most settings of the validation examples *Penta* and *Bucky* are the same (table A.2): the Virial is calculated, all bonds are shaken. A pairlist is constructed every five steps, with a cutoff of 1.4nm, without twin-range interactions. The print block is set to: 200/200/0. *Penta* and *Bucky* differentiate in the following:

- *Penta* (Perturbation example: growing a soft cavity into water): 25,000 time steps are required to observe and verify the perturbation effect. Initial shake and centre of mass motion removal from initial configuration is disabled. A pressure coupling is applied.

  write block: 0 0 0 50 50 1

- *Bucky* (eight bucky C60 balls in water): 12,500 time steps are simulated.

  write block: 250 0 0 250 0 1

Number of Particles	**Truncated Octahedron**		**Rectangular Box**
	with Water (*Thr2*)	**without Water (Thr1)**	**with Water (*ThrG2*)**
solute atoms	3,078	3,078	3,078
solvent atoms	16,281	0	32,883
total (atoms)	19,359	3,078	35,961
solute charge groups	1,285	1,285	1,285
solvent charge groups	5,427	0	10,961
solute molecules	1	1	1
total (charge groups)	6,712	1,285	12,246
simulation box size: x · y · z , [nm]	7.44		6.95 · 7.20 · 7.42

Table A.1 Benchmark systems

Number of Particles	**Rectangular Box**	
	Bucky	**Penta**
solute atoms	480	5
solvent atoms	19,776	2,775
total (atoms)	20,256	2,780
solute charge groups	480	3
solvent charge groups	6,592	925
solute molecules	8	1
total (charge groups)	7,072	928
simulation box size: x · y · z , [nm]	5.62 · 5.95 · 6.09	2.87 · 2.92 · 3.28

Table A.2 Validation systems

B Platforms and Compilation

B.1 Reference machines

- **SunUltra30**: 296MHz CPU with integrated FPU, primary cache 16KB instruction plus 16KB data on-chip, 2MB L2 cache off-chip, 640MB main memory, SunOs 5.5.1 in multi-user mode. Compiler: Sun f77 4.0. Network connection: FastEthernet. For compilation, the following flags applied:

```
FFLAGS = -pg -native -xildoff -O4 -libmil -dalign -unroll=4 -r8 -depend
CFLAGS = -native -pg
```

- **SunUltra10**: 440MHz CPU with integrated FPU, primary cache 16KB instruction plus 16KB data on-chip, 2MB L2 cache off-chip, 512MB main memory, SunOs 5.5.1 in multi-user mode. Compiler: Sun f77 4.0. Network connection: FastEthernet. Compilation: see SunUltra30.

- **IBM SP–2**: distributed-memory message-passing machine. 64 computational nodes: 4 frames with 16 thin nodes each. One control workstation, two frontend systems, 200GB disk subsystem.

  Nodes: 160MHz, POWER2 Super Chip, 4.5GB internal disk, 256MB memory, switch adapter. For compilation, the following flags applied:

```
FFLAGS = -O4 -qEXTNAME -qMAXMEM=4096
CFLAGS = -O4 -Q=100 -qARCH=pwr2 -qMAXMEM=4096 -qTUNE=pwr2 -qUNROLL=4
```

- **SGI Octane**: 295MHz R10000 single processor, 64KB primary cache (32KB data, 32KB instruction), 2MB secondary cache, 128MB main memory. Network connection: FastEthernet, Myrinet. For compilation, the following flags applied:

```
FFLAGS = -align64 -xildoff -LNO:Ou_max=4
CFLAGS = -align64 -xildoff -LNO:Ou_max=4
```

- **Alpha workstations**: 500MHz Alpha processor 21164, 2MB off-chip L3 cache, 128MB main memory. Network connection: FastEthernet, Myrinet. For compilation, the following flags applied:

```
FFLAGS = -fast -u -nofor_main
CFLAGS = -fast -u -unroll 4
```

- **Myrinet** network infrastructure: 1) dual 8-port SAN switch, thus at most 14 workstations might be interconnected by bridging ports A and B on the switch. 2) 32Bit/33MHz PCI to Myrinet SAN adapter, 1MB cache.

B.2 Compiling Parallel GROMOS

Switches

The file *c.h* includes global constants and switches to specify the type (sequential, parallel with barrier sync., etc.) and behaviour (e.g. verbose message passing mode) of the binary executable. Modifications on these settings require recompilation. The major options are shortly described, starting with the default settings for maximum performance (all checks disabled, parallel version with barrier sync.):

- JNBRANGE_CHK: no. Enables additional array range checks during pairlist construction.
- SORT_JNBV: no. If enabled, the pairlist entries are sorted by increasing particle sequence number.
- RANGE_CHK: no. Enables array range checks.
- ALLOC_CHK: no. If enabled: return values of `malloc` are checked.
- MORE_SYNC: yes. Barrier sync. switch
- USE_MPI: yes. Determines the inclusion of the message passing library.
- VERBBOSE_MPI: no. If enabled: prints out all MPI communications to stderr.
- PRTIM: yes. If enabled: timing information is printed to stderr.

Versions

- *promd.ps:* This program is the barrier sync hand-coded parallel version. The settings in file c.h differ from the default as follows:

  -

- *promd.pa:* This program is the hand-coded parallel version without barrier sync. The settings in file c.h differ from the default as follows:

  MORE_SYNC: no
  USE_MPI: yes

- *promd.pg:* This program contains generated code (chapter 5). All communications are blocking and synchronous. Neither broadcast nor collective communication are used. The code generator ignores the switches.
- *promd.64/promd.32:* These programs are based on GROMOS96 fortran source code with the exception of some timing functions written in C. The dot-suffix indicates single (.32) or double (.64) precision floating point arithmetic. Common block constants are set such that all benchmarks can be executed without recompiling.

Support Tools

- transform the pairlist to be GROMOS96 compliant. Export/import the pairlist, compare two pairlists.

C EBNF of the Specification Language

```
specList = "specification" "{" header funcList ifaceDefList declarationList
                               graphDef outMappingDecl "}".

header = "header" "{" fileDefSequence "}".
  fileDefSequence = fileDef {";" [fileDef] }.
    fileDef = CSchedDef | HSchedDef | HInitDef | CInitDef.
      CSchedDef = "CoprocScheduler" "=" ident.
        HSchedDef = "HostScheduler" "=" ident.
          HInitDef = "initHostStatic" "=" ident.
            CInitDef = "initCoprocStatic" "=" ident.

funcList = "functions" "=" "{" string {";" [string] } "}".

ifaceDefList = "interfaces" "{" ifaceDef {";" [ifaceDef] } "}".
  ifaceDef = ident "{" inputDef outputDef "}".
    inputDef = "inputs" ifVarDeclSeq.
    outputDef = "outputs" ifVarDeclSeq.
      ifVarDeclSeq = "{" [ ifVarDecl {";" ifVarDecl} ] "}".
        ifVarDecl = VarDecl "," class.
          class = "dynamic" | "static".

declarationList = "declarations" "{" [declaration] {";" [declaration] } "}".
  declaration = mappingDecl | initSwitch.
    mappingDecl = "mapping" "=" ("AllonC" | "AllbutSonC" |
                                 "AllonA" | "AllbutSonA").
      initSwitch = "init_Dynamic_data" "=" BoolValue.

graphDef = "graph" graphElementList.
  graphElementList = "{" graphElement {";" [graphElement] } "}".
    graphElement = condition | endCondition | function
      condition = condStmt "," boolExp "," boolExp ":" stmtList ":" stmtList.
        boolExp = BoolValue | ident ["." ident].
      endCondition = endcStmt [arrow].
        arrow = "-" ">" stmtList.
      function = funcStmt "," funcType "," inputMapping [arrow].
        funcType = "Host" | "Coprocessor" | "Sequential" | "Parallel".
        inputMapping = "inputs" mapAssignmentList.
          mapAssignmentList = "{" [mapAssignment] {";" [mapAssignment] } "}".
            mapAssignment = ident [":" ident] "=" mapSource [lengthList].
              mapSource = (number | "mpiId" | "mpiNumNodes" | source)
              lengthList = ":" ident { "," ident}.
                source = ident [instNum "." ident].

stmtList = stmt { "," stmt}
  stmt = condStmt | endcStmt | funcStmt.
    condStmt = "cond" instNum.
    endcStmt = "endc" instNum.
    funcStmt = ident instNum.
      instNum = "(" number ")".

outMappingDecl = "outputMapping" "{" ident mapAssignmentList
                                     ident mapAssignmentList "}".

BoolValue = "true" | "false".

varDecl = varType (simpleVarDecl | arrayDecl).
  varType = ident.
  simpleVarDecl = ident.
  arrayDecl = "*" ident.

string = """ {ident | number | specChar} """.
  specChar = die fuer eine Pfadangabe verwendbaren Sonderzeichen

ident = letter {letter | digit | "_"}.
  letter = all characters (small and capital)
  digit = all numbers from 0 to 9
```

D GROMOS96P Specification

D.1 Control Flow Graph

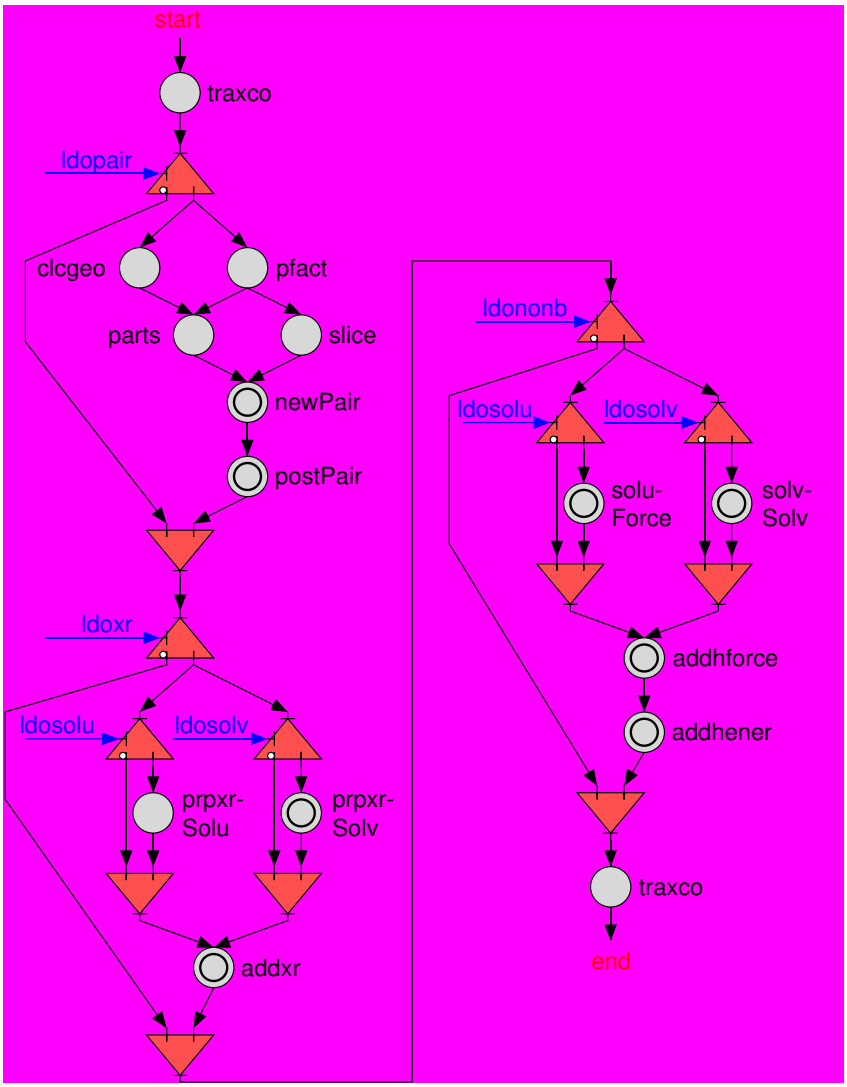


Figure D.1 Complete GROMOS96P control flow specifiaction.

D.2 Control/Data Flow Specification

```
// Gromos96P specification: /md/work/GromosP/Gromos_spec.txt
specification {
  header {
    hostScheduler     = scheduleH;
    coprocScheduler   = scheduleP;
    initHostStatic    = initStaticH;
    initCoprocStatic  = initStaticP;
  }
```

The parallel scheduler running on the host (host scheduler) is named *scheduleH*, the parallel scheduler running on all clients (client scheduler) is named *scheduleP*; similarly, *initStaticH* is the name for the initialization function on the host and *initStaticP* that for the initialization function running on the clients. The giving of names is arbitrary.

```
  functions = {"pfact"; "slice"; "clcgeo"; "parts"; "newpair; "postpair";
   "prpxrSolv"; "prpxrSolu"; "addxr"; "soluForce"; "solvSolv";
   "addhforce"; "addhener"; "traxco_1"; "traxco_2"}
  interfaces {
    scheduleH {
      inputs {
        float *xcoord, dynamic;
        int xcoordl, static;
        float *box, dynamic;
        int boxl, static
        int ldopair, dynamic;
        int ldosolu, static;
        int ldosolv, static;
        int ldoxr, static;
      }
      outputs {
        float *force, dynamic;
        int forcel, dynamic;
        float *vir, dynamic;
        int virl, dynamic;
        float *epel, dynamic;
        int epell, dynamic;
        float *eprf, dynamic;
        int eprfl, dynamic;
        float *eprc, dynamic;
        int seprcl, dynamic;
        float *seplj, dynamic;
        int sepljl, dynamic;
        float *dedlam, dynamic;
        int dedlaml, dynamic;
        float *dedmu, dynamic;
        int dedmul, dynamic;
      }
    };
    scheduleP {
      inputs {
        int ldopair, dynamic;
        int ldosolu, static;
        int ldosolv, static;
        int ldoxr, static;
      }
      outputs {
      }
    }
  }
```

The GROMOS specification declares control flow variables as static, as well as the lengths of both coordinates and box-size arrays. The dynamic data flow consists of atomic coordinates and the simulation box size on the input; on the output atomic forces, energies and the molecular Virial is provided. Note that only control flow variables appear on the interface of *scheduleP*. These must be communicated to the clients before generated code is called. In case of GROMOS, control flow variables do not change during a simulation thus have to be communicated advantageously outside the main loop only once; this task is not part of the specification.

```
declarations {
  mapping     = AllbutSonC;
  init_Dynamic_data = false;
}
graph {
  traxco_1(1), Sequential, inputs {
      x   = scheduleH(0).xcoord;
      xl  = scheduleH(0).xcoordl: xcoord;
      dir = 1;
    } -> cond(1);
  cond(1), scheduleH.ldonot, scheduleP.ldonot: endc(1): clcgeo(1), pfact(1);
  clcgeo(1), Sequential, inputs {
      xcoordl = traxco.xl: x;
      xcoord =  traxco.x;
    } -> parts(1);
  pfact(1), Sequential, inputs {
      boxl = scheduleH(0).boxl: box;
      box =  scheduleH(0).box;
    } -> parts(1), slice(1);
  slice(1), Sequential, inputs {
      nboxl = pfact(1).nboxl: nbox;
      nbox =  pfact(1).nbox;
    } -> newPair(1);
  parts(1), Sequential, inputs {
      xcgl = clcgeo(1).xcgl: xcg;
      xcg =  clcgeo(1).xcg;
      boxl = scheduleH(0).boxl: box;
      box =  scheduleH(0).box;
      pfact = pfact(1).pfact;
      nboxl = pfact(1).nboxl: nbox;
      nbox =  pfact(1).nbox;
    } -> newPair(1);
  newPair(1), Parallel, inputs {
      headl = parts(1).headl: head;
      head  = parts(1).head;
      taill = parts(1).taill: tail;
      tail  = parts(1).tail;
      xcgl  = clcgeo(1).xcgl: xcg;
      xcg   = clcgeo(1).xcg;
      boxl  = scheduleH(0).boxl: box;
      box   = scheduleH(0).box;
      nboxl = pfact(1).nboxl: nbox;
      nbox =  pfact(1).nbox;
      pfact = pfact(1).pfact;
      fslice = slice(1).fslice;
      nslice = slice(1).nslice;
    } -> postPair(1);
  postPair(1), Parallel, inputs {
      inbl  = newPair.inbl: inb;
      inb   = newPair.inb;
      jnbl  = newPair.jnbl: jnb;
      jnb   = newPair.jnb;
      headl = parts(1).headl: head;
      head  = parts(1).head;
      chainl= parts(1).taill: tail;
      chain = parts(1).tail;
      nboxl = pfact(1).nboxl: nbox;
      nbox =  pfact(1).nbox;
      pfact = pfact(1).pfact;
      fslice = slice(1).fslice;
      nslice = slice(1).nslice;
    } -> endc(1);
  endc(1) -> cond(2);
  cond(2), scheduleH.ldoxr, scheduleP.ldoxr: endc(2): cond(3), cond(4);
  cond(3), scheduleH.ldosolu, scheduleP.ldosolu: endc(3): prpxrSolu;
  cond(4), scheduleH.ldosolv, scheduleP.ldosolv: endc(4): prpxrSolv;
  prpxrSolu(1), Sequential, inputs {
      xcoords   = scheduleH(0).xcoord;
      xcoordsl  = scheduleH(0).xcoordl: xcoord;
      nboxl     = pfact(1).nboxl: nbox;
      nbox      =  pfact(1).nbox;
    } -> endc(3);
  prpxrSolv(1), Parallel, inputs {
      headl   = parts(1).headl: head;
      head    = parts(1).head;
      taill   = parts(1).taill: tail;
```

```
    tail    = parts(1).tail;
    xcoord  = scheduleH(0).xcoord;
    xcoordl = scheduleH(0).xcoordl: xcoord;
    nboxl   = pfact(1).nboxl: nbox;
    nbox    =  pfact(1).nbox;
    pfact   = pfact(1).pfact;
    fslice  = slice(1).fslice;
    nslice  = slice(1).nslice;
} -> endc(4);
endc(3) -> addxr(1);
endc(4) -> addxr(1);
addrxr(1), Parallel, inputs {
    xrsl   = prpxrSolu.xrsl: xrs;
    xrs    = prpxrSolu.xrs;
    xrvl   = prpxrSolv.xrvl: xrv;
    xrv    = prpxrSolv.xrv;
} -> endc(2);
endc(2) -> cond(5);
cond(5), scheduleH.ldononb, scheduleP.ldononb: endc(5): cond(6), cond(7);
cond(6), scheduleH.ldosolu, scheduleP.ldosolu: endc(6): soluForce;
cond(7), scheduleH.ldosolv, scheduleP.ldosolv: endc(7): solvSolv;
soluForce(1), Parallel, inputs {
    headl   = parts(1).headl: head;
    head    = parts(1).head;
    chainl  = parts(1).taill: tail;
    chain   = parts(1).tail;
    xcoordl = traxco.xl: x;
    xcoord  = traxco.x;
    nboxl   = pfact(1).nboxl: nbox;
    nbox    = pfact(1).nbox;
    pfact   = pfact(1).pfact;
    fslice  = slice(1).fslice;
    nslice  = slice(1).nslice;
    xrl     = addxr.xrl: xr;
    xr      = addxr.xr;
    boxl    = scheduleH(0).boxl: box;
    box     = scheduleH(0).box;
    inbsl   = postPair.inbsl: inbs;
    inbs    = postPair.inbs;
    jnbsl   = postPair.jnbsl: jnbs;
    jnbs    = postPair.jnbs;
    rmu     = ScheduleH(0).rmuloc;
    rlam    = *p_rlam;
} -> endc(6);
solvSolv(1), Parallel, inputs {
    inbvl   = postPair.inbvl: inbv;
    inbv    = postPair.inbv;
    jnbvl   = postPair.jnbvl: jnbv;
    jnbv    = postPair.jnbv;
    xcoordl = traxco.xl: x;
    xcoord  = traxco.x;
    xrvl    = prpxrSolv.xrvl: xrv;
    xrv     = prpxrSolv.xrv;
    boxl    = scheduleH(0).boxl: box;
    box     = scheduleH(0).box;
    headl   = parts(1).headl: head;
    head    = parts(1).head;
    chainl  = parts(1).taill: tail;
    chain   = parts(1).tail;
    nboxl   = pfact(1).nboxl: nbox;
    nbox    = pfact(1).nbox;
    fslice  = slice(1).fslice;
    nslice  = slice(1).nslice;
} -> endc(7);
endc(6) -> addhforce(1);
endc(7) -> addhforce(1);
addhforce(1), Parallel, inputs {
    soluforce  = soluForce(1).f;
    soluforcel = soluForce(1).fl: soluforce;
    soluvir    = soluForce(1).vir;
    soluvirl   = soluForce(1).virl: soluvir;
    solvforce  = solvSolv(1).f;
    solvforcel = solvSolv(1).fl: solvforce;
    solvvir    = solvSolv(1).vir;
    solvvirl   = solvSolv(1).virl: solvvir;
    ldosolu    = scheduleH(0).ldosolu;
```

```
        ldosolv    = scheduleH(0).ldosolv;
    } -> addhener(1);
    addhener(1), Parallel, inputs {
        sepel      = soluForce(1).eel;
        sepell     = soluForce(1).eell: sepel;
        seprf      = soluForce(1).erf;
        seprfl     = soluForce(1).erfl: seprf;
        seprc      = soluForce(1).erc;
        seprcl     = soluForce(1).ercl: seprc;
        seplj      = soluForce(1).elj;
        sepljl     = soluForce(1).eljl: seplj;
        vepel;     = solvSolv(1).eel;
        vepell     = solvSolv(1).eell: vepel;
        veprf      = solvSolv(1).erf;
        veprfl     = solvSolv(1).erfl: veprf;
        veprc      = solvSolv(1).erc;
        veprcl     = solvSolv(1).ercl: veprc;
        veplj      = solvSolv(1).elj;
        vepljl     = solvSolv(1).eljl: veplj;
        el         = soluForce(1).eglel;
        rf         = soluForce(1).eglrf;
        rc         = soluForce(1).eglrc;
        lj         = soluForce(1).eglljl;
        el34       = soluForce(1).eg34el;
        rf34       = soluForce(1).eg34el;
        rc34       = soluForce(1).eg34el;
        lj34;      = soluForce(1).eg34el;
        ldosolv    = scheduleH(0).ldosolu;
        ldosolu    = scheduleH(0).ldosolv;
      } -> endc(5);
    endc(5) -> traxco_2(1);
    traxco_2(1), Sequential, inputs {
        x:  collect = addhforce(1).gforce;
        xl          = addhforce(1).gforcel: x;
        dir         = 0;
      } -> endc(2);
    endc(1);
  } // end graph
  outputMapping {
    scheduleH {
        force: collect  = traxco_2(1).x;
        forcel          = traxco_2(1).xl: force;
        vir: collect    = addhforce(1).gvir;
        virl            = addhforce(1).gvirl: vir;
        epel: collect   = addhener(1).gepel;
        epell           = addhener(1).gepell: epel;
        eprf: collect   = addhener(1).geprf;
        eprfl           = addhener(1).geprfl: eprf;
        eprc: collect   = addhener(1).geprc;
        seprcl          = addhener(1).gercl: eprc;
        seplj: collect  = addhener(1).gelj;
        sepljl          = addhener(1).geljl: eplj;
        dedlam: collect = addhener(1).gdedlam;
        dedlaml         = addhener(1).gdedlaml: dedlam;
        dedmu: collect  = addhener(1).gdedmu;
        dedmul          = addhener(1).gdedmul: dedmu;
    }
    scheduleP {
    }
  }
} // end specification
```

The GROMOS specification makes use of the collect function principally at the end of the specification, because partial forces and energies are summed up on the host to reconstruct the global data structure. The output mapping of *scheduleP* remains empty, since no result data are used on the clients. For a different decomposition scheme, e.g. domain decomposition, result data would be required on the clients as well.

Bibliography

[1] Allen, M.P. and D.J. Tildesley: *Computer Simulation of Liquids*. Oxford University Press, (1987)

[2] Analog Devices Inc.: *Sharc DSP Microcomputer ADSP-21160 Preliminary Technical Data*. One Technology Way, Norwood MA, USA, (1998)

[3] Analog Devices Inc.: *ADSP-21000 Family. C Tools Manual*. One Technology Way, Norwood MA, USA, (1998)

[4] Belina, F.; D. Hogrefe and A. Sarma: *SDL with Applications from Protocol Specifications*. Prentice-Hall, (1991)

[5] Bekker, H: *Molecular Dynamics Simulation Methods Revised*. PhD thesis, Rijksuniversiteit Groningen, (1996)

[6] Billeter, S.: *Quantum dynamical simulation of non-adiabatic proton transfers in aqueous solution: methodology, molecular models and applications*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Diss. ETH no. 12751, (1998)

[7] Blickle, T: *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Diss. ETH no. 11894, (1997)

[8] Case, D.A.; D:A. Pearlman; J.W. Caldwell; T.E. Cheatham III; W.S. Ross; C.L. Simmerling; T.A. Darden; K.M. Merz; R.V. Stanton; A.L. Cheng; J.J. Vincent; M. Crowley; D.M. Ferguson; R.J. Radmer; G.L. Seibel; U.C. Singh; P.K. Weiner and P.A. Kollman: *AMBER 5*. University of California, San Francisco, (1997)

[9] Cassandras, C.G.: *Discrete Event Systems: Modeling and Performance Analysis*. Aksen Associates Incorporated Publishers, (1993)

[10] Esser, R: *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Diss. ETH no. 11869, (1997)

[11] Haberlandt, R.; S. Fritzsche; G. Peinel and K. Heinzinger: *Molekulardynamik, Grundlagen und Anwendungen*. Vieweg Lehrbuch Physik, (1995)

[12] Hirst, D.M: *A Computational Approach to Chemistry.* Blackwell Scientific Publications, (1990)

[13] Hockney, R.W. and J.W. Eastwood: *Computer Simulation using Particles*. IOP publishing Ltd., (1988)

[14] Klein, R: *Algorithmische Geometrie*. Addison Wesley, (1997).

[15] Gajski, D.D; F. Vahid; S. Narayan and Jie Gong: *Specification and Design of Embedded Systems*. Prentice Hall, (1994)

[16] Gajski, D.; N. Dutt; A. Wu and S. Lin: *High-Level Synthesis and System Design*. Kluwer Academic Publishers, (1992)

[17] Gupta, R.K: *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, (1995)

[18] Harel, D. and M. Politi: *Modeling Reactive Systems with Statecharts: The Statemate Approach*. i-Logix inc. (1997)

[19] Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, (1985)

[20] Kneubühl, F.K.: *Repetitorium der Physik*. Teubner Studienbücher Physik, 1994

[21] Pachero, P.S.:*Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc, San Francisco, CA, (1997)

[22] Scott, W.R.P.: *Molecular Dynamics Simulation of Biomolecules: Software Engineering, Parallelisation and Methodology.* PhD thesis, Eidgenössische Technische Hochschule Zürich, Diss. ETH no. 12603, (1998)

[23] Thomas D. and P. Moorby: *The Verilog Hardware Description Language*. Kluwer, (1991)

[24] Wolfram, S.: *The MATHEMATICA Book*. Cambridge University Press, Third Edition, (1996)

[25] Silberschatz, A.: *Operating Systems Concepts*. Addison-Wesley, Reading, MA. (1994)

[26] van Gunsteren, W.F.; S.R. Billeter; A.A. Eising; P.H. Hünenberger; P. Krüger; A.E. Mark; W.R.P. Scott and I.G. Tironi: *Biomolecular Simulation: The GROMOS™96 Manual and User Guide*. Hochschulverlag vdf AG an der ETH Zürich, Switzerland, (1996)

[27] Alexandrov, A; M. F. Ionescu; K. E. Schauser and C. Scheiman: *LogGP: Incorporating Long Messages into the LogP model. One step closer towards a realistic model for parallel computation*. Tech Report TRCS95, Department of Cimputer Science, University of California Santa Barbara, http://www.cs.ucsb.edu/, (1995)

[28] Banicescu, I. and Rong Lu: *Experiences with Fractiling in N-Body Simulations*. Proceedings of High Performance Computing '98, pp. 121–126, (1998)

[29] Bekker, H.; H.J.C. Berendsen: *GROMACS: A parallel computer for molecular dynamics simulation*. Conference proceedings, Physics Computing '92, (1992)

[30] Berendsen, H.J.C; D. van der Spoel and R. van Drunen: *GROMACS: A message-passing parallel molecular dynamics implementation*. Computer Physics Communication 91, pp. 43-56, (1995)

[31] Berendsen, H.J.C. and J Mavri: *Quantum dynamics simulation of a small quantum system embedded in a classical environment*. In: Quantum mechanical Simulation Methods for studying Biological Systems, Proceedings of Les Houches Workshop, Springer-Verlag, pp. 157–179, (1996)

[32] Billeter, S.R and W.F. van Gunsteren: *A modular molecular dynamics/quantum dynamics program for non-adiabtic proton transfers in solution*. Computer Physics Communication 107, pp. 61-97, (1997)

[33] Boden, N.J.; D. Cohen; R.E.Felderman; A.E.Kulawik; C.L. Seitz; J.N. Seitzovic and Wen-King Su: *Myrinet: A Gigabit per Lecond Local Area Network*. IEEE Micro, vol.15, no. 1, pp. 29-36, (1995)

[34] Brooks, B.R; R.E. Bruccoleri; B.D. Olafson; D.J. States; S. Swaminathan and M. Karplus: *CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations*. Journal of Computational Chemistry, no. 4, pp. 187-217, (1983)

[35] Car, R and M. Parrinello: *Unified Approach for Molecular Dynamics and Density Functional Theory.* Phys. Rev. Lett. 55, 2471 (1986)

[36] Clark, T.W.; R.v. Hanxleden; J.A. McCammon and L.R. Scott: *Parallelising Molecular Dynamics using Spatial Decomposition*. Proceedings of the Scalable High Performance Computing Conference '94, Knoxville, TN, (1994)

[37] Culler, D; R.M. Karp; D. Patterson; A. Sahay; K.E. Schauser; E.E. Santos; R. Subramonian and T. von Eicken: *LogP: Towards a Realistic Model of Parallel Computation*. In proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego CA, (1993)

[38] Everaers, R and K. Kremer: *A fast grid search algorithm for molecular dynamics simulations with short-range interactions*. Computer Physics Communication 81, pp. 19–55, (1994)

[39] Fortune, S. and J.C. Wyllie: *Parallelism in random access machines*. In proceedings of the 10th Annual Symposium on Theory of Computing, (1978)

[40] Fukushige, T.; M. Taiji; J. Makino; T. Ebisuzaki and D. Sugimoto: *A Highly-parallelised Special-purpose Computer for Many-body Simulations with an Arbitrary Central Force: MD-GRAPE*. The Astrophysical Journal, no. 468, pp. 51–61

[41] Gerber, M. and T. Gössi: *Automatic Parallelisation of GROMOS96™ Molecular Dynamics Simulation Package for Distributed Memory Environments*. Proceedings of High Performance Computing '99, pp. 25–30, (1999)

[42] Gerber, M: *A new Gromos™ Benchmark on SunUltra30/60, DEC Alpha21164, IBM RS/6000 SP.* Internal Paper, http://www.tik.ee.ethz.ch/~md

[43] Gerber, M and T. Gössi: *Project HAM: Hardware Moved Molecules, Annual Report 1997.* TIK-Report Nr. 38, Computer Engineering and Networks Laboratory, ETH Zürich, (1998)

[44] Gerber, M. and T. Gössi: *Parallel Coprocessor Architectures for Molecular Dynamics Simulation: A Case Study in Design Space Exploration*. Proceedings of the 1998 IEEE International Symposium on Circuits and Systems ISCAS98, (1998)

[45] Gössi, T: *Computer Platforms for Molecular Dynamics Simulation*. PhD thesis, Eidgenössische Technische Hochschule Zürich, (expected 2000)

[46] IEEE: *IEEE Standard VHDL Language Reference Manual, Std. 1076*. IEEE Press, New York, (1987)

[47] Kufrin, R. and S. Subramaniam: *UHGROMOS at the National Centre for Supercomputing Applications*, (1998)

[48] Lee, E.A. and D.G. Messerschmitt: *Synchronous Dataflow*. Proceedings of the IEEE, 75(9): 1235–1245, (1987)

[49] Lomdahl, P.S. and D.M. Beazley: *Multi-Million Particle Molecular Dynamics on MPPs*, Second International Workshop, PARA95, Lyngby, Denmark, (1995)

[50] Lomdahl, P.S.; P. Tamayo; N. Grønbech-Jensen and D.M. Beazley: *50 GFlops Molecular Dynamics on the Connection Machine 5*. Proceedings of Supercomputing '93, IEEE Computer Society, pp. 520–527, (1993)

[51] Luty, B.A.; M.E. Davis; I.G. Tironi and W.F. van Gunsteren: *A Comparison of Particle-Particle, Particle-Mesh and Ewald Methods for calculating electrostatic Interactions in periodic Molecular Systems*. Molecular Simulation, vol. 14, pp. 11–20, (1994)

[52] Makino, J.: *Stellar Dynamics on 200 Tflops Special-Purpose Computers: Grape-6*. International Symposium on Supercomputing, (1997)

[53] Mojtabaeezamani, G.: *SP POWER3 SMP Node System Architecture*. IBM white paper, (1999)

[54] Müller-Plathe, F.; W. Scott; and W.F. van Gunsteren: *PARALLACS: a benchmark for parallel molecular dynamics*. Computer Physics Communication, no. 384, pp. 102–114, (1994)

[55] Nyland, L.; J. Prins; R.H. Yun; J. Hermans; H.C. Kum and L. Wang: *Achieving Scalable Parallel Molecular Dynamics Using Dynamic Spatial Domain Decomposition Techniques*. Journal of Parallel and Distributed Computing, no. 47, pp. 125–138, (1997)

[56] Pant, A.: *A High Performance MPI Implementation on the NTSC VIA Cluster*. National Center for Supercomputing Applications (NCSA), University of Ilinois at Urbana-Champaign, (1999)

[57] Pearlman, D.A.; D.A. Case; J.W. Caldwell; W.S. Ross; T.E. Cheatham III; S. DeBolt; D. Ferguson; G. Seibel and P. Kollman: *AMBER, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules*. Comp. Phys. Communication, no. 91, pp. 1-41, (1995).

[58] Roccatano, D.; R. Bizzari; G. Chillemi; N. Sanna and A. di Nola: *Development of a Parallel Molecular Dynamics Code on SIMD Computers: Algorithm for use of a Pair List Criterion*. Journal of Computational Chemistry, vol. 19, no. 7, pp. 685–694, (1998)

[59] Ruf, L and T. Strösslin: *Molekülsimulation aif IBM RS 6000 SP*. Semester thesis, Computer Engineering and Networks Lab (TIK), Eidgenössische Technische Hochschule Zürich (ETH), (1999)

[60] Scott, W.R.P.; P.H. Hünenberger; I.G. Tironi; A.E. Mark; S.R. Billeter; J. Fennen; A.E. Torda; T. Huber; P. Krüger and W.F. van Gunsteren: *The GROMOS Biomolecular Simulation Program*. Journal of Physical Chemistry A, no. 103, submitted

[61] Sivasubramiam, A.; A. Singla; U. Ramachandran and H. Venkateswaran: *An Application-Driven Study of Parallel System Overheads and Network Bandwidth Requirements*. IEEE Transactions on parallel and distributed systems, vol. 10, no. 3, 193–210, (1999)

[62] Taylor, V.E.; R. L. Stevens and K. E. Arnold: *Parallel Molecular Dynamics: Implications for Massively Parallel Machines*. Journal of Parallel and Distributed Computing, no. 45, pp. 166–175, (1997)

[63] Toyoda, S.; H. Miyagawa; K. Kitamura; T. Amisaki; E. Hashimoto; H. Ikeda; A. Kusumi and N. Miyakawa: *Development of MD Engine: High-Speed Accelerator with Parallel Processor Design for Molecular Dynamics Simulations*. Journal of Computational Chemistry, vol. 20, no. 2, pp. 185–199, (1999)

[64] van Gunsteren, W.F.; H.J.C. Berendsen; F: Colonna; D: Perahia; J.P. Hollenberg and D. Lellouch: *On Searching Neighbours in Computer Simulation of Macromolecular Systems*. Journal of Computational Chemistry, vol. 5, no. 3, pp. 272–279, (1984)

[65] Wirth, N: *Compilerbau*. Teubner Studienbücher Informatik, (1986)

Curriculum Vitae

Personal

Name:	Martin Gerber
Date of Birth:	15th December 1969
Place of Birth:	Liestal, Switzerland
Citizen of:	Langnau i.E., Switzerland

Education

	Primary and Secondary School in Pratteln, Switzerland
1988	Matura Typus C, Gymnasium Liestal
1994	Dipl. El.-Ing. ETH
1994–1996	Research assistant in the research group of Prof. Dr. A. Kündig in the Computer Engineering and Networks Laboratory at the ETH Zürich
1996–1999	Ph.D. student in the research group of Prof. Dr. L. Thiele in the Computer Engineering and Networks Laboratory at the ETH Zürich