University of Alberta

SIMULATION OF FEMTOSECOND LASER ABLATION OF SILICON

by

Roman Holenstein 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science.**

Department of Electrical and Computer Engineering

Edmonton, Alberta

Spring 2005

University of Alberta

Library Release Form

Name of Author: Roman Holenstein

Title of Thesis: Simulation of Femtosecond Laser Ablation of Silicon

Degree: Master of Science

Year this Degree Granted: 2005

Date: Nov. 12, 2004

"Alright brain, you don't like me, and I don't like you. But let's just do this, and I can get back to killing you with beer."

Homer Simpson

Abstract

Femtosecond laser ablation is an important process in the micromachining and nanomachining of microelectronic, optoelectronic, biophotonic and MEMS components. The process of laser ablation of silicon is being studied on an atomic level using molecular dynamics (MD) simulations. We investigate ablation thresholds for Gaussian laser pulses of 800 nm wavelength, in the range of a few hundred femtoseconds in duration. Absorption occurs into a hot electron bath which then transfers energy into the crystal lattice. The simulation box is a narrow column 5.4 nm×5.4 nm×81 nm with periodic boundaries in the x and y transverse directions and a 1-D heat flow model at the bottom coupled to a heat bath to simulate an infinite bulk medium corresponding to the solid bulk material. A modified Stillinger-Weber potential is used to model the silicon atoms. The calculated ablation thresholds of silicon are compared to values reported in experimental and theoretical studies. We obtain reasonable agreement with experiment for pulse lengths of 100 fs and 200 fs (1/e) giving thresholds of 0.13 J/cm^2 and 0.19 J/cm^2, respectively. The ablation threshold is found to have a square-root dependence on the pulse length.

Acknowledgements

I would like to thank my supervisors Dr. Robert Fedosejevs and Dr. Ying Y. Tsui for the opportunity to pursue my M.Sc. at the University of Alberta. I am very grateful for your support and guidance. This has been a valuable learning experience for me.

I wish to express my gratitude to my fellow graduate students, colleagues, and friends. Special thanks go to Michael Argument, Michael Cummings, Sean Kirkwood, Matthew Reid, and Michael Taschuk, you have been an invaluable source of expertise.

I would also like to thank Dr. Eleanor E.B. Campbell for providing an initial version of a molecular dynamics code which was used as a starting basis for this work.

I would like to acknowledge financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), the Informatics Circle of Research Excellence (iCore), and the Canadian Institute for Photonic Innovations (CIPI). And I would like to express my appreciation to the WestGrid project for providing computational support.

And I would like to especially thank my parents, my brothers and my sister for all their love and continuous support, for their constant encouragement and for always being there for me. Thank you!

Contents

List of Figures

List of Tables

Nomenclature

$\vec{r}$	a vector
$\lvert \vec{r} \rvert$	length of vector $\vec{r}$
$\vec{r}_{ij}$	a vector from particle i to particle j: $\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$
r_{ij}	distance between particle i and particle j: $r_{ij} = \lvert \vec{r}_{ij} \rvert$
F_{abs}	absorbed fluence
F_{inc}	incident fluence
λ	wavelength
τ_L	laser pulse length
c	speed of light
e	Coulomb constant
E_g	band gap energy
$h, \hbar$	Planck's constant ($\hbar = h/2\pi$)

k_B Boltzmann's constant

m_{Si} mass of silicon atom

m_{el} mass of electron

m_h mass of hole

T_{el} electron temperature

T_{ph} lattice (phonon) temperature

a-Si amorphous silicon

AFM atomic force microscopy

BASH Bourne-Again SHell

c-Si crystalline silicon

CN coordination number

density functional theory

EAM embedded-atom method

HF heat flow

MD molecular dynamics

MEMS microelectromechanical systems

MPI Message Passing Interface

PA plasma annealing

PBC periodic boundary condition

PM photomultiplier

QMS quadrupole mass spectrometer

Si silicon

SW Stillinger-Weber (interaction potential)

UV ultraviolet (light)

Chapter 1

Introduction

Computer simulations have become a vital part in research. The use of computers has allowed us to solve problems that were previously too complex to do. In many cases, a lot of assumptions and approximations were necessary in order to solve any real world problem. Now many problems can be solved numerically. Computer simulations are not only used to solve a particular problem, but also to test theoretical models. These models are implemented in a computer simulation, and the results from it can then be compared to results obtained from an equivalent physical experiment. Since the first implementation of a computer only five decades ago, the computer speeds and memory capacities have increased at an exponential rate, however, there are still limitations. Many complex or computationally intensive problems may now be solved in a reasonable amount of time. But efficient algorithms are still required in order to tackle such problems, particularly using realistic parameters.

There are various types of computer simulations; the one that has been developed in this thesis

is based on *molecular dynamics* (MD). Other related examples include Monte Carlo simulation and molecular mechanics. In MD, a system of particles is studied by computing the interactions between the particles and integrating their paths based on Newtonian mechanics.

Molecular dynamics has found many applications. It was first employed by B.J. Alder and T.E. Wainwright to study liquids (using hard spheres) [AW57]. Defects in crystals have also been studied using MD simulations [MB91, MB93, KUO00]. Further examples of where MD simulations have been applied include studies in fracturing, surface physics, friction, clustering, biomolecules, and electronic and dynamical properties of materials [LLR+88, AB87, MB92, MB93, IMM+98, Lee98, ZGB99, VC00, RKL+02, NBG+03, HMM04].

Silicon is an important material in industry. Many devices are fabricated from Si, such as micro-chips or MEMS devices for example. Ultrafast laser are often used in the production of these devices. The purpose of this project was to implement a molecular dynamics simulation of silicon (Si) to simulate the process of laser ablation. In laser micromachining ultrafast laser pulses, pico- to femto-seconds in duration, are used to remove small amounts of material from a substrate, for example to create holes or micro-structures. An understanding of the mechanism underlying the ablation process is important in order to improve the micromachining quality. Molecular dynamics is a convenient tool to study this process, as it provides a microscopic view of the material, which would be very difficult if not impossible to obtain experimentally. Also, it allows for the isolation of subsystem processes so they can be studied independently and their contribution to the overall process assessed.

1.1 Micro-/nano-machining

Laser ablation is the physical process exploited in laser micromachining. The various applications of this include fabrication and repair of MEMS (microelectromechanical systems) devices and optical devices, and in creating via holes, cutting electrical paths, isolating features for electrical microchips, and direct writing of microfluidic systems.

MEMS devices consist of micron-sized movable parts. In the standard fabrication of these devices some leftover material may remain on the device, preventing proper operation. This excess material can be removed using laser ablation.

Laser ablation can also be employed in the fabrication of microfluidic devices. These devices are used for example in DNA and protein analysis and microchemical analysis. Generating these devices may require vertical micron sized holes to be drilled into the substrate (generally some type of glass). This can be accomplished using ultrafast laser ablation. Currently the challenge is in avoiding cracks and inhomogeneities surrounding the hole. A better understanding of the ablation process, in particular for femto-second pulses, may help to improve the drilling techniques.

1.2 Femtosecond laser ablation

In femtosecond laser ablation multiphoton absorption plays an important role. Through multiphoton absorption it is possible to excite electrons even if the bandgap is larger than the energy of a single photon. The combined energy of two photons may allow a transition to take place in that case. The excess energy (if any) will add to the kinetic energy of the excited electron.

The excited electrons are susceptible to the electric field of the incident laser pulse, gaining

kinetic energy in the process (inverse Bremsstrahlung). These highly energetic carriers then knock out electrons from neighbouring atoms and excite them to the conduction band (impact ionization).

1.3 Laser ablation parameters

Material is removed from a substrate by ablating it using one or more laser pulses. The ablation process, and thus the features created, depends on the properties of the laser pulse(s) and the material used. The following subsections will briefly outline the dependence of the ablation process on the laser pulse width, wavelength, and material, as well as some of the physical processes involved in laser ablation.

1.3.1 Pulse width

The features created in the target material significantly depend on the length of the laser pulse incident on the target material. As the laser pulse irradiates the material, it deposits energy at the focal spot. This creates a temperature gradient and the heat diffuses to a length scale L given below, resulting in surface cracks and other unwanted damage to the target material. The diffusion length is given by

$$L = 2\sqrt{\tau D} \tag{1.1}$$

where τ is the pulse length and D is the thermal diffusivity of the target material. For example Silicon has a thermal diffusivity of $0.8\,\mathrm{cm}^2/\mathrm{s}$, thus using a 1 ns pulse the diffusion length is 566 nm, while for a 1 fs pulse the diffusion length is only 0.57 nm. Thus one expects much cleaner holes to be created using a femtosecond laser pulse compared to a much longer nanosecond pulse. This

has been confirmed in experiment. The threshold fluence is also dependent on the pulse length [BBK+02].

1.3.2 Wavelength

Different wavelengths get absorbed differently in a particular material. Thus the ablation process will be affected by the choice of wavelength. For example in most materials ultraviolet (UV) light is generally more strongly absorbed than light at longer wavelengths. Thus the skin depth for UV is very short, allowing ablation at low energies. The reflectivity also depends on the wavelength, and for silicon is higher in the UV and drops to about 30% in the near-IR. Figure 1.1 shows the dependence of the absorption coefficient and the reflectivity on the wavelength. Further, the focal spot size depends on the wavelength used – the shorter the wavelength, the smaller the focal spot that can be achieved, allowing for smaller features to be created.

1.3.3 Number of shots

Experiments have indicated that incubation effects have significant impact in multishot laser ablation, especially in the femtosecond regime. It has been seen that in irradiating with several laser pulses, each being below the single shot threshold for ablation, at the substrate, that after a few shots ablation does take place, even though the individual pulses do not have sufficient energy to ablate the material in a single pulse [BBK+02].

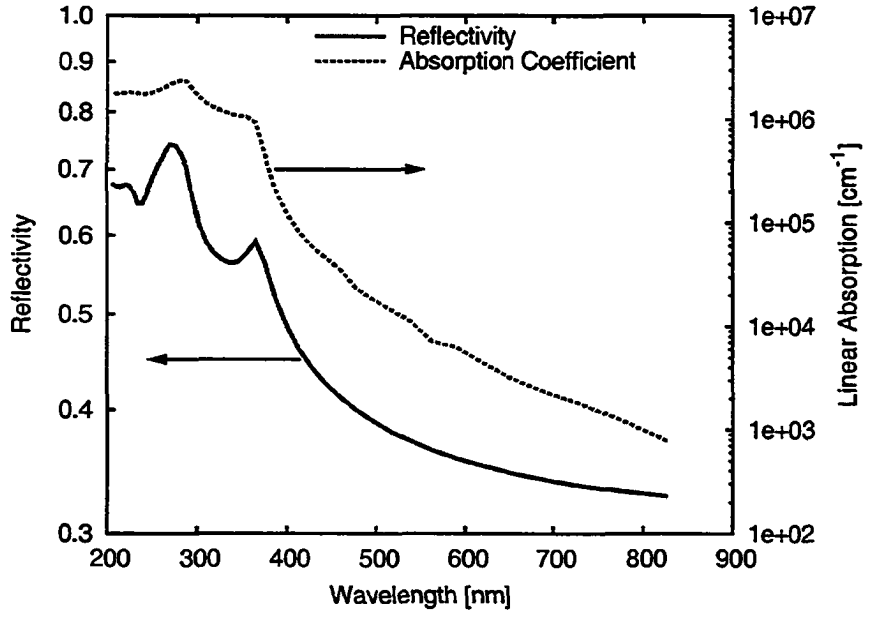


Figure 1.1: Reflectivity and linear absorption coefficient of silicon [AS83]

1.4 Simulation models

In the next few sections we will briefly introduce some of the models that have been used in simulating laser ablation.

1.4.1 Heat diffusion model

For the study of laser ablation, the heat diffusion model offers a simple approach. The energy of the laser added to the system and the heat is diffused into the bulk according to the temperature gradient driven heat flow (HF). Generally this model has validity when using longer pulse lengths (ns time scale). This is due to the fact that recombination of excited electrons with the ions in the

lattice occurs on a much shorter time scale and equilibrium (or at least quasi-equilibrium) can be assumed in this case. However, a recent study has employed this technique to investigate ultrafast laser ablation of metals, dielectrics and semiconductors [BSR+04] and obtained reasonable agreement with experimental results. In our project we use a heat flow model at the bottom of the MD simulation volume to simulate an infinite bulk medium. Chapter 2 gives a detailed overview of the heat diffusion model.

1.4.2 Hydrodynamic model

The hydrodynamic model simulates the system as a fluid (or gas). This is particularly useful in studying laser interaction with plasmas. Since no distinct particles are considered, this model is independent of scale and the computational effort mainly depends on the desired accuracy. Therefore this approach is very efficient and allows for the study of various phenomena from laser-plasma interactions to modelling of plasma in the earths magnetosphere. In covalent materials (e.g. Si, glass, ...), the particles are very close together and tightly bound (significant interaction between the atoms), therefore the system cannot be treated as a gas or fluid and thus this model is not appropriate for this project. However, this approach has been used in modelling of laser ablation of metals [ACD+01, TLFC00].

1.4.3 Particle models

There are various techniques that can be employed to model an N-body system. Among them are Monte Carlo (MC), particle-in-cell (PIC), and molecular dynamics (MD). In PIC, the potential surface is computed on a grid and the forces acting on the particles are interpolated from the grid points.

This method is particularly useful in modelling fluids or gases of charged particles in electromagnetic fields [Wei77]. In MD the forces on the particles are calculated directly for each particle. While PIC and MD are deterministic models, MC employs statistics and random sampling to obtain the state of system for the next time step.

1.5 State of the art

There has been a significant amount of research done in the modelling of laser ablation for pulse lengths of millisecond to nanosecond in duration [Zha01, LLW+97, PMK94, Bäu00]. Laser interaction and ablation using longer pulses, particularly for metals, can be reasonably well described using heat flow models, since the interaction is dominated by thermal processes [Bäu00]. With recent advances in ultrafast laser ablation a lot of effort has been put towards modelling and understanding the ablation process in the short-pulse regime.

In ultrafast laser ablation, the dominant processes underlying the removal of matter are different than for short pulses. The pulse length in fs-ablation is on the same order of magnitude or shorter than the time it takes for the electrons to thermalize (10^{-14}s) and to reach equilibrium with the lattice (10^{-12}s) [Aga84]. This requires at least a two-temperature model, i.e. separate electron system from lattice, and at high fluences a kinetic model. Most simulations of femtosecond and picosecond laser ablation of silicon (and other semiconductors) employ molecular dynamics [HGC98, WIOM00, LLM01], and some use MD in conjunction with Monte Carlo techniques [ZGZ02, LLM03]. A few have used a two-temperature heat diffusion model to study the ultrafast laser interaction with silicon [PDDS95, BSR+04].

1.5.1 Interaction potentials

Various groups developed models to describe the interaction between the particles in solid silicon [SW85, Ter88, DC98, BH85]. Balamane et al. have compared several empirical interaction potentials for silicon [BHT92] and concluded that none of the six potentials stands out as superior. They all have their strengths and limitations. E.R. Cowley [Cow88] arrived at a similar conclusion by calculating the elastic constants and selected normal-mode frequencies for the Stillinger-Weber (SW) [SW85], Tersoff [Ter88], and Biswas-Hamann [BH85] potentials and comparing the results with experiments, but found that overall the SW potential gives the best description of the lattice dynamics. For the study of laser ablation the Stillinger-Weber potential is predominantly used (see section 3.1.4). The embedded-atom method (EAM) [Bas87, BNW89, Bas92] has also been used to study silicon, for example in modelling of crack propagation [SBN89]. A German group has also used *ab initio* MD [CP85, AKPF94] to look at laser melting of Si [SAPF96, SAPF97].

There are also several interaction models available for the study of glass, in particular for fused silica [TMTM88, vBKvS90]. The one most often used in modelling glass is the BKS potential, developed by Beest, Kramer and Santen [vBKvS90]. The BKS potential was developed from ab initio calculations on small clusters and experimental data.

Zhigilei et al. have concentrated on simulating laser ablation of organic solids. Also using molecular dynamics, they employ a so-called "breathing sphere" model, which allows them to model an entire molecule as a single particle, where the particle properties are dependent on its internal structure. In this way it is possible to speed up the simulation and allows examination of larger volumes of the solid [ZKG97]

1.5.2 Silicon threshold studies

There are a few groups studying the ablation process by pico- and femtosecond laser pulses for silicon. Following we will briefly present some of the key contributions that are relevant to this project.

A group in Montréal (Canada), headed by Michel Meunier, has studied the mechanisms involved in the ablation of Si for picosecond pulses in the near UV [LLM00a, LLM00b, LLM01, LLM03]. They employ a MD thermal annealing model in which the relaxation of the carriers occurs by transfer of kinetic energy to the lattice by spontaneous emission of optical phonons. This model applies for carrier densities below the critical value of $n_c \approx 10^{22}\,\mathrm{cm}^{-3}$ for silicon and pulse lengths down to about 10 ps. The atomic interactions are modelled using the SW potential. They deduce that in this regime one-photon interband transition is the dominant laser absorption mechanism and ignore multi-photon and free-carrier absorption. Following the absorption an electron-hole (e-h) pair is generated and the valence counter of the exited atom decremented by one. The relaxation mechanisms taken into account in their model are carrier-phonon scattering and carrier diffusion. In their latest work, which considers pulse durations down to $\tau_L = 500\,\mathrm{fs}$, they have added a Monte-Carlo model to account for relaxation of hot electrons and holes through a cascade of scattering events. By following the thermodynamic trajectory of the system in the temperature-density plane, they identified phase explosion as the primary ablation mechanism under near-adiabatic cooling conditions near the threshold for shorter pulses ($< 10\,\mathrm{ps}$) [LLM03]. For longer pulses or under non-adiabatic cooling conditions it was found that fragmentation due to pressure buildup was the only relevant ablation mechanism [PL02, LLM03]. At $\lambda = 266\,\mathrm{nm}$ they obtained ablation thresholds

(incident fluence) of 0.35 J/cm^2 ($\tau_L = 500$fs) and 0.54 J/cm^2 ($\tau_L = 50$ps), and 0.30 J/cm^2 ($\tau_L = 10$ps) for a wavelength of $\lambda = 308$nm.

While most simulations of this type use periodic boundary conditions (PBC) in the transverse direction to the incident laser pulse, Herrmann et al. used a different approach [HGC97, HGC98]: a cylindrical rim surrounds the system and acts as a heat bath, i.e. the particles in that shell of a few Angstrom in thickness are damped to account for the heat dissipation into the bulk. To make this setup computationally feasible the laser spot size was shrunk to 25 Å in diameter and the absorption coefficient increased by a factor of 2000. The SW potential is used to model the atomic interaction. Upon absorption of a photon by an atom, it is marked as excited. The resulting change in the potential was implemented by randomly breaking a number of bonds matching the degree of excitation. For picosecond and femtosecond pulses they observe that the main material removal occurs within a few picoseconds. The ablation thresholds obtained are between 3 J/cm^2 and 8 J/cm^2 for pulse lengths in the range of 10 fs up to 5 ps (threshold increasing with pulse length). This is much higher than the experimental values reported [CSTB$^+$99], probably due to excessive cooling by the boundary "wall" (cylindrical rim) since the system size is relatively small (only 100 Å in diameter).

Ohmura et al. have also investigated laser ablation of silicon using MD and the SW potential. They found that a Si(111) surface more easily evaporates than a Si(100) under laser irradiation ($\tau_L = 200$fs, $\lambda = 266$) and obtained qualitative agreement with experiment [IWF$^+$98]. The effective surface bond density (in thermal equilibrium) for Si(111) is larger than for Si(100) and therefore more energy is absorbed near the surface of Si(111) as compared to Si(100). The shock wave

velocity was measured for both crystal orientations and good agreement with elastic theory was obtained, showing that the shock wave propagates faster in the Si[111] direction (9.36km/s) than in the Si[100] direction (8.44km/s) [WIOM00].

Using *ab initio* molecular dynamics based on density functional theory (DFT)in lieu of the commonly used SW potential, Alavi, Parrinello, and Frenkel have investigated laser heating of silicon [SAPF96]. As suggested in the "plasma annealing" (PA) model [VTSH79, VTS79], under intense laser irradiation (short pulses on order of 100 fs) the semiconductor material can be rapidly driven into a disordered state (melting). This has also been shown in experiment [SYH83b, SYH83a]. According to the PA, a high level of electronic excitation can lead to a weakening of the bonds. Alavi's *ab initio* model was developed to describe this situation [AKPF94, SAPF96]. Their simulations were able to reproduce the fast melting. However, the ions did not remain cool as the PA model predicts and instead reach temperatures at around the melting point of Si ($T_m = 1680$K). The liquid formed in this process had different properties than for normal liquid Si, such as higher coordination number (11-13) and a high diffusion coefficient.

Jeschke et al. employ MD simulations based on an electronic tight-binding Hamiltonian [JGL+02]. The model takes into account the nonequilibrium created in the electronic system due to irradiation with an intense laser pulse. Using periodic boundary conditions in all three dimensions of the MD supercell containing 64 atoms, and applying a constant external pressure (10^5 Pa), the melting and ablation process was studied for pulse durations of 20 fs and 500 fs (Gaussian). It was found that the increase in ablation threshold from $\tau_L = 20$fs to $\tau_L = 500$fs was 67%, which agrees with the trend found by Bonse et al. [JGL+02].

In order to reduce the computational cost and thus allowing larger systems and longer time frames to be considered, Zhigilei et al. developed a breathing-sphere model which is used to study organic solids [ZG99b, ZKG97, ZG99c, ZG99a]. In this model, a group of atoms (e.g. a molecule) is represented by a single particle, the size of which changes according to the internal degree of freedom. The internal dynamics of such a particle is approximated with an appropriate potential. Their model has shown that ejection due to phase explosion and the relaxation of laser induced pressure are the primary ablation mechanisms [ZG00].

1.6 Layout of the thesis

A simulation code has been developed to model the process of laser ablation. E.B. Campbell provided the initial version of the code [HGC98]. We have made some modifications in an effort to improve the simulation and results. In the initial code the Si bulk had dimensions $100\text{Å}(x) \times 100\text{Å}(y) \times 50\text{Å}(z)$, with the laser incident from the top ($+z$). The particles were surrounded by a cylindrical rim of 6 Å thickness containing atoms that were thermally coupled to a heat reservoir (see fig. 1.2). The system layout has been modified to periodic boundary conditions in the transverse direction to the laser pulse, and a heat flow model has been added at the bottom of the MD system to provide a more realistic coupling to the heat bath. Melting and ablation thresholds for pulses in the range of a few hundred femtosecond in duration have been obtained. The results are in reasonable agreement with experiment, particularly for 100 fs and 200 fs (width at 1/e). This is a significant improvement over the initial code, which gave thresholds that were one order of magnitude too high.

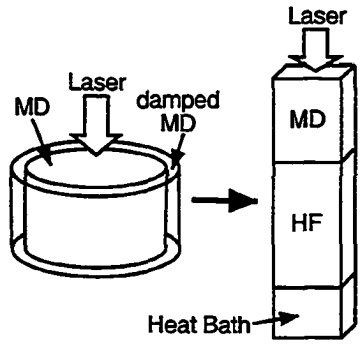


Figure 1.2: Simulation setup of the initial (left) and current (right) code.

Chapter 2 gives a detailed description of the heat diffusion model used to simulate the semi-infinite bulk. Chapter 3 describes the molecular dynamics technique and the interatomic forces describing the material. It also presents some computational optimizations that were implemented in an effort to speed up the simulations. The coupling between the molecular dynamics and heat flow system is outlined in chapter 4. The laser absorption process is presented in chapter 5. The results and discussions are given in chapter 6 and conclusions and future directions are outlined in chapter 7.

Chapter 2

Heat-Flow Model

We employ both a heat flow (HF) and a molecular dynamics (MD) model to simulate the laser-matter interaction and ablation process of silicon. In the near-infrared the linear absorption length (skin depth) is on the order of a few microns, which is very large for a molecular dynamics simulation and computationally expensive. In order to cut this cost we attach a one-dimensional (1-D) HF model to the bottom of the MD system. With this hybrid system we can simulate the ablation process in the MD region while coupling the energy to the Si bulk through the HF region. The MD system can then be reduced to less than one micron and the simulation is extended far enough (few times the absorption length) with the less costly HF model.

This chapter will outline the heat flow model. Both one-temperature (1-T) and two-temperature (2-T) models have been used in the simulations. The description below will focus on the 2-T model. The 1-T model is then simply obtained by removing the electron system and absorbing the energy from the laser directly in the lattice.

2.1 Heat-flow equations

In a solid, heat is transfered by conduction. The energy flux depends on the temperature gradient and some material specific coefficient (thermal conductivity), as given by the Fourier heat conduction law [Bai99]:

$$(\text{net energy flux}) = -\kappa \nabla T \tag{2.1}$$

where κ is the thermal conductivity and T is the temperature. Now consider a small volume V_{small}. The change in temperature can be expressed in terms of its heat capacity and the net energy flux out of that volume. Equating the increase in energy in the volume with the energy flowing into the volume gives:

$$C_P \frac{\partial T}{\partial t} = -\int_{\text{surface over } V_{\text{small}}} (-\kappa \nabla T) \cdot d\vec{A} \tag{2.2}$$

where $d\vec{A}$ is a vector denoting a small surface area and normal to the surface pointing outward, and C_P is the heat capacity of that volume. Dividing by V_{small} we get the time-dependent Fourier equation:

$$c_P \frac{\partial T}{\partial t} = \nabla (\kappa \nabla T) \tag{2.3}$$

c_P is the specific heat capacity per unit volume. This equation assumes that the heat flow is over length scales much longer than the mean free path of the particles. For a crystalline solid the mean free path is on the order of the atomic spacing. The nearest neighbour distance for Si at room temperature is 2.35 Å, so for the Fourier equation to hold we must have length scales much larger than that.

We can now express heat flow in the lattice and electron system using equation 2.3 and couple

them by adding an energy transfer term [Bäu00]:

$$\left\{ \begin{array}{rcl} c_{el}\frac{\partial T_{el}}{\partial t} & = & \nabla(\kappa_{el}\nabla T_{el}) - g(T_{el} - T_{ph}) + Q \\ c_{ph}\frac{\partial T_{ph}}{\partial t} & = & \nabla(\kappa_{ph}\nabla T_{ph}) + g(T_{el} - T_{ph}) \end{array} \right\} \tag{2.4}$$

where the subscripts *ph* and *el* denote the lattice (phonon) and electron gas, respectively, c_{ph} and c_{el} are the specific heat capacities (per unit volume), T is the temperature, Q is the energy entering the system (laser), and the function g is the energy transfer coefficient. For crystalline silicon, the thermodynamic properties are given in table 2.1.

	symbol	crystalline	amorphous
density $[g/cm^3]$	ρ	2.32	2.32
heat capacity $[J/(gK)]$	c_p	0.71	0.8
thermal conductivity $[W/(cmK)]$	κ	1.5	0.018

Table 2.1: Thermodynamic coefficients for crystalline (c-Si) and amorphous (a-Si) silicon at 300 K ([Bäu00], pg. 697).

Further silicon parameters are summarized in [PDDS95] and [BHT92]. Narayan et al. give a scaling law for the thermal conductivity: $\kappa(T) = 1585/T^{1.229} W/(cmK)$ for $300K < T < 1370K$ [NC92]. The coupling factor g is taken to be [Aga84]

$$g = \frac{3 n_{el} k_B}{\tau_c} \tag{2.5}$$

where τ_c is the energy relaxation time and n_{el} is the number density of the carriers [Aga84, FP96]. $\tau_c = 1\,\text{ps}$ was used to match the MD system (see section 5.4).

2.2 Electron gas

The electron system in a semiconductor has to be modelled differently than for metals. The density of electrons in the conduction band changes significantly at different temperatures. Also, the absorption of laser energy will promote electrons into the conduction band. Following we will first look at the density due to thermally exited electrons and then look at the "creation" of conduction band electrons due to laser absorption (sec. 2.2.2).

2.2.1 Thermally excited electrons

To get the concentration of intrinsic carriers, we follow the approach given by Kittel [Kit96] (pg. 216). Assuming simple parabolic band edges, the Fermi-Dirac distribution gives the following electron concentration in the conduction band:

$$n_{th} = 2\left(\frac{m_{el}k_BT_{ph}}{2\pi\hbar^2}\right)^{3/2}\exp\left(\frac{\mu - E_c}{k_BT_{ph}}\right) \tag{2.6}$$

with the chemical potential μ given by ([Kit96], pg. 220, eqn. 47):

$$\mu = E_c - \left[\frac{1}{2}E_g + \frac{3}{4}k_BT_{ph}\ln\left(\frac{m_h}{m_{el}}\right)\right] \tag{2.7}$$

where m_{el} and m_h are the electron and hole mass, respectively, and E_g is the band gap energy, as indicated in figure 2.1. For crystalline silicon, the band gap energy is $E_g = 1.17\,\text{eV}$ at 0 K and $E_g = 1.11\,\text{eV}$ at 300 K.

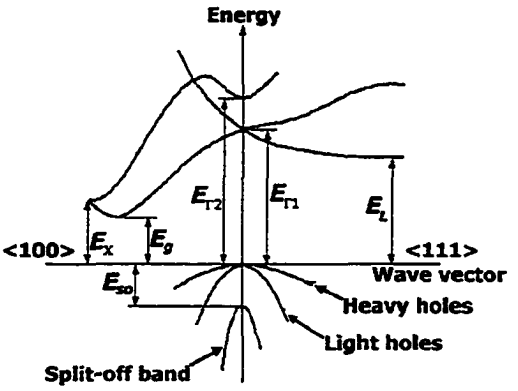


Figure 2.1: Band structure of silicon at 300 K [Iof].

Now, (for simplicity) assuming that $m_{el} = m_h$, and substituting equation 2.7 into eqn. 2.6, we get:

$$n_{th} = 2\left(\frac{m_{el}k_BT_{ph}}{2\pi\hbar^2}\right)^{3/2}\exp\left(-\frac{E_g}{2k_BT_{ph}}\right) \tag{2.8}$$

At a lattice temperature of 300 K, the concentration is on the order of 10^{10}cm^{-3}, which is only a very small fraction of the valence electrons ($\approx 2\times10^{23}\text{cm}^{-3}$).

2.2.2 Laser absorption

The photons can be absorbed by the electrons in two ways: photo-excitation of electrons and inverse bremsstrahlung. Since the density of thermally excited electrons is much smaller than the density of valence electrons, photo-excitation is the dominant process. Given a photon energy of $E_p = hc/\lambda$ and bandgap of $E_g = 1.11\,\text{eV}$ at a lattice temperature of 300 K, the kinetic energy of the photoexcited

electron is $E_k = E_p - E_g$. This gives an electron temperature of

$$T_e = \frac{2}{3}\frac{E_k}{k_B} = \frac{2}{3k_B}\left(\frac{hc}{\lambda} - E_g\right) \tag{2.9}$$

For simplicity we assume no dependence of E_g on the $\vec{k}$-vector, which allows us to avoid phonon absorption in the excitation of the electron to the conduction band. For photons at a wavelength of 760 nm we get an electron temperature of approximately 4000 K.

Exciting electrons into the conduction band of course increases the electron density. Thus the newly gained electrons must be added to the electron model and the temperatures equilibrated (locally). Considering only one-photon absorption, the increase of electron density is equal to the photon density, thus

$$\Delta n_{el} = \frac{\varepsilon_L}{E_p} \tag{2.10}$$

where ε_L is the locally deposited laser energy density given by the laser fluence and Beer's law:

$$\varepsilon_L = \Delta t\, I_L \alpha e^{-\alpha z} \tag{2.11}$$

where Δt is the integration time step, I_L is the laser intensity on entering the system (at $z = 0$), α is the linear absorption coefficient.

In a short pulse, we can assume that no or few of the photo-excited electrons will return to the valence band before the end of the laser pulse. Thus the peak density of photo-excited electrons is:

$$n = \frac{1}{E_p}\int I_L \alpha e^{-\alpha z} dt = \frac{F\alpha}{E_p} e^{-\alpha z} \tag{2.12}$$

where F is the laser fluence. Thus, for a fluence of $0.5\,\mathrm{J/cm^2}$ we expect an electron density of $\approx 2\times10^{21}\mathrm{cm}^{-3}$ at $z=0$, which is about 1% of the valence electrons available.

2.2.3 Heat capacity

The specific heat capacity of the electron gas is given by [Kit96] (pg. 155), which was obtained from a free electron gas model in three dimensions:

$$c_{el} = \frac{1}{2}\pi^2 n k_B \frac{T_{el}}{T_F} \tag{2.13}$$

where T_F is the Fermi temperature, which is directly proportional to the Fermi energy and is given by $T_F = \varepsilon_F / k_B$:

$$\varepsilon_F = \frac{\hbar^2}{2m_{el}} \left(3\pi^2 n\right)^{2/3} \tag{2.14}$$

However, equation 2.13 is only valid for a cold electron gas, i.e. for temperatures $T \ll T_F$. At densities on the order of $10^{21}\mathrm{cm}^{-3}$ the Fermi temperature is about 3000 Kelvin, which is of the same order of magnitude as the expected temperature of the photo-excited electrons. Thus, we cannot use equation 2.13.

At low densities and high temperatures (on the order of T_F or greater), we may assume little interaction between the electrons and treat them as free particles. Thus the heat capacity is [Kit96]

$$c_{el} = \frac{3}{2} n k_B \tag{2.15}$$

This is the electron heat capacity used in our simulations.

2.2.4 Thermal conductivity

The thermal conductivity coefficient κ is defined with respect to the steady-state flow of heat along a temperature gradient [Kit96]:

$$j_U = -\kappa \frac{dT}{dx} \tag{2.16}$$

We can approximate the thermal conductivity coefficient for electrons (κ_{el}) from the one for the lattice and scale it according to the temperature as follows ([Bäu00] p. 271):

$$\kappa_{el} = \kappa_{ph} \frac{T_{el}}{T_{ph}} \tag{2.17}$$

The lattice heat conductivity (κ_{ph}) is given in table 2.1.

2.3 Numerical solution

2.3.1 Discrete formulation

In the following we will lay out the discretized system of equation. Each of the two one-dimensional sub-systems (lattice and electrons) is subdivided into $N+2$ cells, as shown in figure 2.2, where the first and last cell serve to implement the boundary conditions. The time increments $\Delta t \equiv t^{i+1} - t^i$ are assumed to be constant and sufficiently small such that the energy fluxes (in particular the energy transfer between cells) are approximately constant and arbitrarily close to their values at any intermediate time $t^{i+\theta}$ in that interval $[t^i, t^{i+1}]$, with

$$t^{i+\theta} \equiv t^i + \theta \Delta t = (1-\theta) t^i + \theta t^{i+1}, \quad 0 \leq \theta \leq 1 \tag{2.18}$$

The isometric, specific heat coefficient (c_S) is assumed to be constant, while the heat conductivity (κ) may vary with temperature. The density may also change from one time step to the next (generation of free electrons and subsequent recombination). The heat transfer for each of the two sub-systems can be expressed in the following discrete problem [AS92]. The subscripts denote spatial position (cell index) and superscripts denote time:

initial values:
$$T_n^0 = T_{\text{init}}(z_n),\ n = 1,...,N \tag{2.19a}$$
$$E_n^0 = E(T_n^0),\ n = 1,...,N \tag{2.19b}$$

boundary condition at top:
$$q_{1/2}^{i+\theta} = \frac{T_0^{i+\theta} - T_1^{i+\theta}}{R_{1/2}^{i+\theta}},\ \text{with } R_{1/2}^{i+\theta} = \frac{\frac{1}{2}\Delta z}{\kappa_1^{i+\theta}} \tag{2.19c}$$

boundary condition at bottom:
$$q_{N+1/2}^{i+\theta} = \frac{T_{N+1}^{i+\theta} - T_N^{i+\theta}}{R_{N+1/2}^{i+\theta}},\ \text{with } R_{N+1/2}^{i+\theta} = \frac{\frac{1}{2}\Delta z}{\kappa_N^{i+\theta}} \tag{2.19d}$$

interior values:
$$E_n^{i+1} = E_n^i + \frac{\Delta t}{\Delta z}\left[q_{n-1/2}^{i+\theta} - q_{n+1/2}^{i+\theta}\right],\ n = 1,...,N \tag{2.19e}$$

where E is the thermal energy, T is the temperature, κ is the heat conductivity, c_S is the specific heat coefficient (for solid), and R is the thermal resistivity. q denotes the energy flux between the cells and is given by

$$q_{n-1/2}^{i+\theta} = -\frac{T_n^{i+\theta} - T_{n-1}^{i+\theta}}{R_{n-1/2}^{i+\theta}},\ \text{with } R_{n-1/2}^{i+\theta} = \frac{\Delta z}{2}\left(\frac{1}{\kappa_{n-1}^{i+\theta}} + \frac{1}{\kappa_n^{i+\theta}}\right),\ n = 2,...,N \tag{2.19f}$$

and the temperature is (assuming a weak dependence of the specific heat on temperature):

$$T_n^i = \left(T_n^{i-1} + \frac{E_n^i - E_n^{i-1}}{\rho_n^{i-1} c_S}\right) \times \frac{\rho_n^{i-1}}{\rho_n^i} \tag{2.19g}$$

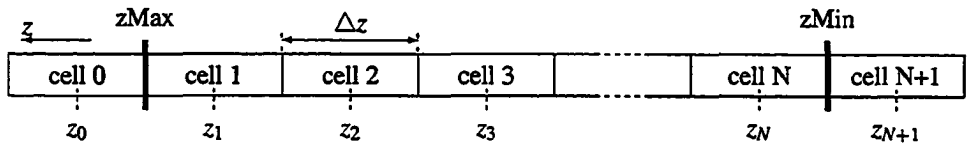


Figure 2.2: Layout of heat flow model

2.3.2 Explicit scheme

Choosing $\theta = 0$ in (2.19), the fluxes are evaluated at the old time step and the new energies and temperature can be evaluated directly. This is the so-called explicit scheme. The numerical integration algorithm is given as follows [vAB95] (pg. 173):

$$\begin{aligned} W_n^{[el]i+1} &= W_n^{[el]i} + \frac{\Delta t}{\Delta z^2}\left\{\kappa_{n,n+1}^{[el]i}\left(T_{n+1}^{[el]i} - T_n^{[el]i}\right) - \kappa_{n-1,n}^{[el]i}\left(T_n^{[el]i} - T_{n-1}^{[el]i}\right)\right\} + Q_n^{[el]i}\Delta t \\ W_n^{[ph]i+1} &= W_n^{[ph]i} + \frac{\Delta t}{\Delta z^2}\left\{\kappa_{n,n+1}^{[ph]i}\left(T_{n+1}^{[ph]i} - T_n^{[ph]i}\right) - \kappa_{n-1,n}^{[ph]i}\left(T_n^{[ph]i} - T_{n-1}^{[ph]i}\right)\right\} + Q_n^{[ph]i}\Delta t \end{aligned} \tag{2.20}$$

with the conductivity between cells n and $n+1$ given by

$$\kappa_{n,n+1}^{i} = 2\left(\frac{1}{\kappa_n^i} + \frac{1}{\kappa_{n+1}^i}\right)^{-1} \tag{2.21}$$

$W^{[el]}$ and $T^{[el]}$ denote the electron energy density and temperature, and $W^{[ph]}$ and $T^{[ph]}$ denote the lattice energy density and temperature, respectively. i is the current time step and n is the cell index, Δt is the time interval and $\Delta z \equiv z_{n+1} - z_n$ is the cell size. The heat entering or exiting the system (to/from lattice or electron gas) is contained in $Q_n^{[ph]i}$ and $Q_n^{[el]i}$ for lattice and electron gas, respectively.

The change in energy gives rise to a temperature change. Since the density may also change,

the new temperature must be scaled accordingly and is given by:

$$T_n^{i+1} = \left(T_n^i + \frac{W_n^{i+1} - W_n^i}{\rho c_p} \right) \times \frac{\rho_n^i}{\rho_n^{i+1}} \tag{2.22}$$

where c_p is the specific heat capacity for the lattice or electron gas.

Since the system is integrated discretely, there is an inherent error introduced into the simulation. If this error grows too fast, the simulation becomes unstable and the results unphysical. This error grows with time step size. Of course if the time steps are too small then rounding errors become significant and may also make the results invalid. We can get an estimate for a good time step size using the stability condition (Courant-Friedrich-Lewy), which is given by [CFL67]:

$$\Delta t < \frac{1}{2} (\Delta z)^2 \min\left[\frac{\rho c_p}{\kappa} \right] \tag{2.23}$$

2.3.3 Implicit scheme

Instead of setting $\theta = 0$, we choose $0 < \theta \leq 1$ with $q^{i+\theta} = \theta q^{i+1} + (1-\theta) q^i$. This results in a system of equations for the temperatures $T_1^{i+1}, \ldots, T_N^{i+1}$.

$$q_{n-1/2}^{i+\theta} = \theta \frac{T_{n-1}^{i+1} - T_n^{i+1}}{R_{n-1/2}^{i+1}} + (1-\theta) \frac{T_{n-1}^i - T_n^i}{R_{n-1/2}^i}, \quad n = 1, \ldots, N \tag{2.24}$$

Now substituting (2.24) into (2.19e) gives:

$$E_n^{i+1} - \frac{\theta\Delta t}{\Delta z}\left[\frac{1}{R_{n+1/2}^{i+1}}T_{n+1}^{i+1} - \left(\frac{1}{R_{n+1/2}^{i+1}} + \frac{1}{R_{n-1/2}^{i+1}}\right)T_n^{i+1} + \frac{1}{R_{n-1/2}^{i+1}}T_{n-1}^{i+1}\right] \tag{2.25}$$

$$= E_n^i + \frac{(1-\theta)\Delta t}{\Delta z}\left[\frac{1}{R_{n+1/2}^{i}}T_{n+1}^{i} - \left(\frac{1}{R_{n+1/2}^{i}} + \frac{1}{R_{n-1/2}^{i}}\right)T_n^{i} + \frac{1}{R_{n-1/2}^{i}}T_{n-1}^{i}\right] \tag{2.26}$$

The right-hand side of (2.26) contains only known values (i.e. all quantities are evaluated at the old time step t^i). For convenience, we shall denote it by b_n^i:

$$b_n^i = E_n^i + \frac{(1-\theta)\Delta t}{\Delta z}\left[\frac{1}{R_{n+1/2}^{i}}T_{n+1}^{i} - \left(\frac{1}{R_{n+1/2}^{i}} + \frac{1}{R_{n-1/2}^{i}}\right)T_n^{i} + \frac{1}{R_{n-1/2}^{i}}T_{n-1}^{i}\right] \tag{2.27}$$

Due to the non-linearity in the system (the thermal resistivity depends on temperature), one cannot solve (2.26) directly (e.g. Gauss elimination), but must be solved using an iterative method. We shall employ a very simple and convenient algorithm, namely the Gauss-Seidel iteration, to determine the energies and temperatures for the next time step. The idea is to solve the n-th equation for the n-th unknown using the latest values of all other variables.

The iteration starts with an initial "guess" for the temperatures and energies, which is set to the values at the old time step (t^i). In order to formulate the algorithm, we shall use a superscript (p) to denote the p-th iteration. We now need to solve the following equation for $E_n^{(p+1)}$ and $T_n^{(p+1)}$:

$$E_n^{(p+1)} + \frac{\theta\Delta t}{\Delta z}\left[\frac{1}{R_{n+1/2}^{(p)}} + \frac{1}{R_{n-1/2}^{(p)}}\right]T_n^{(p+1)} = b_n^i + \frac{\theta\Delta t}{\Delta z}\left[\frac{1}{R_{n+1/2}^{(p)}}T_{n+1}^{(p)} + \frac{1}{R_{n-1/2}^{(p)}}T_{n-1}^{(p+1)}\right] \tag{2.28}$$

where $E_n^{(p+1)}$ and $T_n^{(p+1)}$ are connected by

$$T_n^{(p+1)} = T_n^{i'} + \frac{E_n^{(p+1)} - E_n^{i'}}{\rho_n^{i'} c_S} \tag{2.29}$$

Note that due to excitation of electrons from laser light absorption the density may change in time. In order to accommodate for this, we change the density and scale the temperature accordingly at the beginning of the time step. This is directly followed by adding or removing energy coming in or going out of the system by means other than heat conduction (e.g. due to laser absorption or coupling between lattice and electron system) and then adjusting the temperature again accordingly. The primed superscript (i') denotes the parameter values after this adjustment has taken place, but before the heat conduction has been calculated.

Now, to solve (2.28), we shall simplify the equation by making the following substitutions:

$$\xi_n^{(p)} = \frac{\theta \Delta t}{\Delta z} \left[\frac{1}{R_{n+1/2}^{(p)}} + \frac{1}{R_{n-1/2}^{(p)}} \right] \tag{2.30}$$

$$\phi_n^{(p)} = b_n^i + \frac{\theta \Delta t}{\Delta z} \left[\frac{1}{R_{n+1/2}^{(p)}} T_{n+1}^{(p)} + \frac{1}{R_{n-1/2}^{(p)}} T_{n-1}^{(p+1)} \right] \tag{2.31}$$

Then we can write (2.28) as

$$E_n^{(p+1)} + \xi_n^{(p)} T_n^{(p+1)} = \phi_n^{(p)}, \quad n = 1, \ldots, N \tag{2.32}$$

Now substituting (2.29) into (2.32) and solving for $E_n^{(p+1)}$ gives

$$E_n^{(p+1)} = \frac{\phi_n^{(p)} + \xi_n^{(p)} \left(\frac{E_n^{i'}}{\rho_n^{i'} c_S} - T_n^{i'} \right)}{1 + \frac{\xi_n^{(p)}}{\rho_n^{i'} c_S}} \tag{2.33}$$

Now that we have $E_n^{(p+1)}$ we can compute the temperature from (2.29).

2.3.4 Optimization

Since the electrons diffuse heat much quicker than the lattice, the number of cells for each of the two subsystems differ. For simplicity an integral number of lattice cells , typically about 2-3, are used per electron cell, which generally are a few nanometers in size. Additionally, smaller time steps are used for the electron system than for the lattice. For each lattice time step there are on the order of 100 time steps for the electron system. The heat transfered from the lattice to the electrons is added to a buffer and then gradually added to the electron system at each time step. Similarly for the reverse process.

To further reduce the computation time, an energy buffer can also be put in place of the electrons system, which will absorb the laser energy and transfer it to the lattice on (approximately) the same time scale as the electron system would (same g-parameter). This is identical to having no heat diffusion in the electron system, but it does account for the time delayed heating of the lattice to match the molecular-dynamics system. The system is thereby simulated as a 1-temperature system. This is sufficient for our purposes as the HF model is primarily used as an extension of the MD system and accounts for the rising temperature at the bottom MD-boundary and the for the energy

penetrating the long absorption skin depth.

Chapter 3

Molecular dynamics

Molecular dynamics is a widely used technique in computational physics to simulate molecular-scale models of matter. It was first developed in the 1950s and started gaining widespread attention in the mid-1970s, when computers became powerful and affordable.

Essentially molecular dynamics numerically solves an N-body problem. The trajectories of each particle are computed using Newtonian physics. Knowing the position and velocity of a particle at a given time (and at previous time steps for increased accuracy), as well as the net force acting on the particle, it is possible to compute the position and velocity for the next time step.

Molecular dynamics is a very convenient technique for modelling atomic systems, since it relies only on the underlying interactions between the particles. The difficulty, of course, lies in knowing precisely what the interactions are.

3.1 Basic model equations and assumptions

As with other types of simulations, MD is built on a model. This model describes the movements and interactions between the particles in the simulation. In molecular dynamics simulations, the interaction between particles are described by potentials. Two possible candidates for a potential are described in sections 3.1.3 and 3.1.4. The movement of the particles in response to the given potential(s) are described in the following section.

3.1.1 Newtonian mechanics

Molecular dynamics uses classical, or Newtonian, physics to describe the equations of motion for the particles in the system. Newton's second law provides the acceleration of a particle for a given force acting on it:

$$m_i \ddot{\vec{r}}_i = \vec{f}_i \tag{3.1}$$

where m_i is the mass, $\ddot{\vec{r}}_i$ is the acceleration, i.e. the second derivative with respect to time of the spatial location, and f_i is the force acting on the ith particle. The force can be obtained from the overall potential energy ϕ_i:

$$\vec{f}_i = -\nabla_{\vec{r}_i} \phi_i(\vec{r}_1, \vec{r}_2, ..., \vec{r}_n) \tag{3.2}$$

where $\nabla_{\vec{r}_i}$ denotes the gradient operator with respect to the location of particle i.

3.1.2 Hard and soft spheres

There are two broad classes of molecular dynamics simulations: soft spheres and hard spheres [Rap95, Hai97]. In the case of hard spheres the interaction potential is a simple step function of the following form:

$$u(r_{ij}) = \begin{cases} \infty, & r_{ij} \leq 2R \\ 0, & r_{ij} > 2R \end{cases} \tag{3.3}$$

where r_{ij} is the distance between particles i and j, and R is the radius of the particles. This is for the case of all particles having the same size. In modelling hard spheres, it is more convenient to use kinematics of collisions, rather than employing equation 3.2 using the step function given above (equation 3.3). In collision kinematics conservation of momentum and kinetic energy is used to provide the phase-space trajectories. A detailed description of this is given in chapter 3 of [Hai97].

In the case of soft spheres, the potentials are smooth (at least up to a certain cutoff radius, see section 3.2.3). This was the case for the molecular dynamics simulation of this project, thus a soft sphere model was used. The potentials implemented in the simulation code are the well known Lennard-Jones potential and the Stillinger-Weber potential. These are described in the following sections.

3.1.3 Lennard-Jones potential

The Lennard-Jones (LJ) potential is a widely used potential used to describe the interactions between particles in liquids and gases. It consists of an attractive and a repulsive part. At short distances, the repulsive part dominates, whereas at long ranges the attractive part of the potential is dominant.

The potential is given by [Hai97, Rap95]

$$u_{LJ}(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{3.4}$$

where ε and σ are the energy and length parameters, respectively, specific to the interacting particles in the system. The potential has a minimum at $r_{min} \approx 1.122\sigma$. The shapes of the potential and resulting force obtained from equations 3.2 and 3.4, are shown in figure 3.1. The force due to the LJ potential is given by

$$\vec{f}_{LJ}(\vec{r}_{ij}) = -24\varepsilon\left[2\left(\frac{\sigma^{12}}{r_{ij}^{13}}\right) - \left(\frac{\sigma^{6}}{r_{ij}^{7}}\right)\right]\hat{r}_{ij} \tag{3.5}$$

with $\vec{r}_{ij} \equiv \vec{r}_j - \vec{r}_i$, $r_{ij} = |\vec{r}_{ij}|$ and $\hat{r}_{ij} = \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|}$. $\vec{r}_i$ and $\vec{r}_j$ denote the atomic positions of the interacting particles.

3.1.4 Stillinger-Weber potential

The Stillinger-Weber (SW) potential describes the interactions between Silicon particles and was first proposed by Frank H. Stillinger and Thomas A. Weber in 1985 [SW85] and has since been used in many studies, including the modelling of laser ablation [HGC97, WIOM00, LLM01].

$$\phi = \sum_{i<j} V_2(r_{ij}) + \sum_{i<j<k} V_3(\vec{r}_i, \vec{r}_j, \vec{r}_k) \tag{3.6}$$

where r_{ij} is the distance between two atoms of index i and j, and $\vec{r}_i,\vec{r}_j,\vec{r}_k$ are the positions of atoms i, j, and k, respectively.

$$V_2(r) = \begin{cases} (A_1 r^{-p} - A_2 r^{-q}) \exp\left[\frac{\sigma}{r-a}\right], & r < a \\ 0, & r \geq a \end{cases} \tag{3.7}$$

$$V_3(\vec{r}_i,\vec{r}_j,\vec{r}_k) = h(r_{ij},r_{ik},\theta_{jik}) + h(r_{ji},r_{jk},\theta_{ijk}) + h(r_{ki},r_{kj},\theta_{ikj}) \tag{3.8}$$

where θ_{jik} is the angle between the vectors $\vec{r}_j - \vec{r}_i$ and $\vec{r}_k - \vec{r}_i$. The function h is given as follows:

$$h(r_{ij},r_{ik},\theta_{jik}) = \lambda \exp\left(\frac{\gamma\sigma}{r_{ij}-a}\right) \exp\left(\frac{\gamma\sigma}{r_{ik}-a}\right) \left(\cos\theta_{jik} + \frac{1}{3}\right)^2 \tag{3.9}$$

Note that $\cos(109.47°) = -1/3$, thus h sees a minimum at $\theta_{jik} = 109.47°$. Figure 3.1 shows the pairwise potential energy of the Stillinger-Weber potential in reduced units (i.e. in terms of ε for energy and σ for distance). The general shape is similar to the Lennard-Jones potential. Note that in this figure no three-body interactions are present, i.e. only $V_2(r)$ is shown. Figure 3.2 shows the SW potential in the [100] plane as seen by a particle in a diamond crystal lattice. The values of the constants introduced above are given in table 3.1. Details on the force calculation are provided in appendix A.

3.1.5 Coulomb potential

The interaction between charged particles is governed by the Coulomb potential. Unlike the LJ or SW potential, it is either only attractive or repulsive, depending on two interacting particles. It is

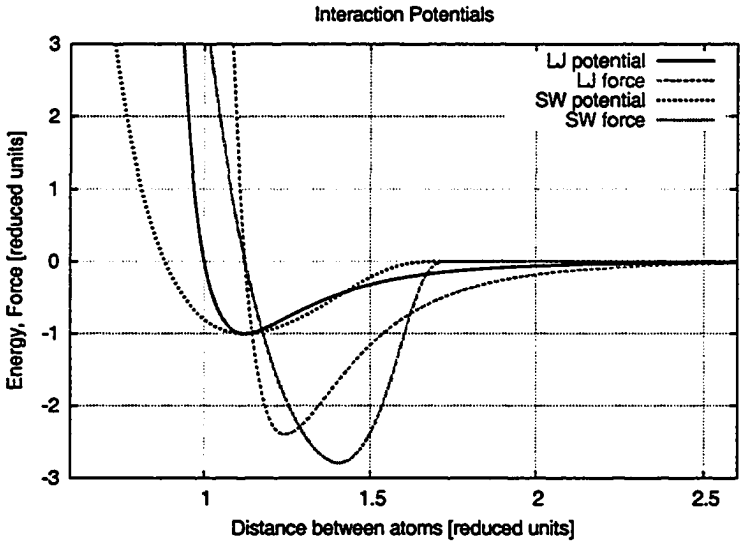


Figure 3.1: Stillinger-Weber potential for two-particle system. Shown is the reduced pair potential as a function of particle separation. The energy is given in multiples of ε and the distance in multiples of σ.

also long-ranged compared to the other two, which adds to the computational cost. The Coulomb potential is given by [Gri99]:

$$u_C(r_{ij}) = -\frac{1}{4\pi\varepsilon_0}\frac{q_i q_j}{r_{ij}} \tag{3.10}$$

where r_{ij} is the separation distance between particles i and j, q_i and q_j are the charges of particles, and ε_0 is the permitivity of free space. The force acting on particle i due to particle j is

$$f_C(\vec{r}_{ij}) = \frac{1}{4\pi\varepsilon_0}\frac{q_i q_j}{r_{ij}^2}\hat{r}_{ij} \tag{3.11}$$

3.1.6 Lattice construction

Crystalline silicon has a diamond structure. A diamond lattice (fig. 3.3) consists of two face centered

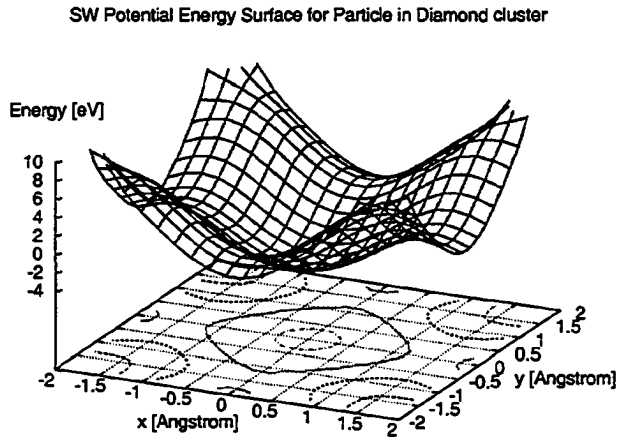


Figure 3.2: SW potential energy in the [100] plane of a diamond lattice. The equilibrium position is at coordinate $(0,0)$ and the nearest neighbour distance is 2.35Å.

cubic (FCC) structures, which are offset by $a\sqrt{3}/4$, where a is the length of the unit cell. Figure 3.4 shows the structure of an FCC lattice. Thus to construct a diamond lattice, one can simply construct two FCC lattices and offset them by $\vec{r} = a/4[\hat{x}+\hat{y}+\hat{z}]$. As illustrated on the right-hand side of figure 3.4, an FCC lattice can be constructed by placing four atoms in a unit cell and then filling the lattice volume with these unit cells.

3.1.7 Limitations

Molecular dynamics simulations deal with atomic scale particles. How can we then justify using classical physics to describe motions of the particles? Is not the physics at this scale governed by quantum mechanics and would require us to use Schrödinger's equation? In order to test the validity of using classical approximations, we can look at the de Broglie thermal wavelength, which is given

	Stillinger and Weber [SW85]	Balamane et al. [BHT92]
A_1 [eV Å^4]	177.29442232	189.360881
A_2 [eV]	15.27991263	16.31972277
λ [eV]	45.512028	48.61499998
a [Å]	3.77118	3.77118
σ [Å]	2.0951	2.0951
p	4	4
q	0	0
γ	1.20	1.20

Table 3.1: Parameters for the Stillinger-Weber potential

as follows:

$$\Lambda = \sqrt{\frac{2\pi\hbar^2}{Mk_BT}} \tag{3.12}$$

As long as $\Lambda \ll a$, where a is the mean nearest neighbour distance, then the classical approximation is valid. In liquids, the ratio Λ/a is approximately 0.1 for Lithium and Argon and on the order of 0.2 for Silicon, and the ratio decreases for heavier particles. For very light elements such as Hydrogen, Helium, or Neon, the classical approximation is no longer valid and quantum mechanics must be used.

3.2 Numerical method

This section will outline the numerical methods used for the path integration of the particles. There are various algorithms for this, two classes are leap-frog type methods and predictor-corrector methods. Verlet's algorithm falls into the first category, and Gear's algorithm is of the latter type.

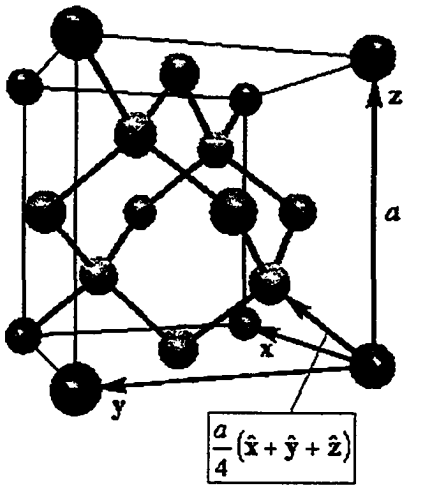


Figure 3.3: Diamond crystal structure

3.2.1 Verlet's algorithm

Verlet's algorithm is based on the Taylor expansion of the coordinate variable $\vec{r}$:

$$\vec{r}(t+\Delta t) = \vec{r}(t)+\dot{\vec{r}}(t)\Delta t+\frac{1}{2}\ddot{\vec{r}}(t)\Delta t^2+\frac{1}{6}\dddot{\vec{r}}(t)\Delta t^3+O(\Delta t^4) \quad (3.13)$$

$$\vec{r}(t-\Delta t) = \vec{r}(t)-\dot{\vec{r}}(t)\Delta t+\frac{1}{2}\ddot{\vec{r}}(t)\Delta t^2-\frac{1}{6}\dddot{\vec{r}}(t)\Delta t^3+O(\Delta t^4) \quad (3.14)$$

where Δt is a small time step. Adding and subtracting the above equations yields:

$$\vec{r}(t+\Delta t) = 2\vec{r}(t)-\vec{r}(t-\Delta t)+\ddot{\vec{r}}(t)\Delta t^2+O(\Delta t^4) \quad (3.15)$$

$$\dot{\vec{r}}(t) = \frac{\vec{r}(t+\Delta t)-\vec{r}(t-\Delta t)}{2\Delta t}+O(\Delta t^2) \quad (3.16)$$

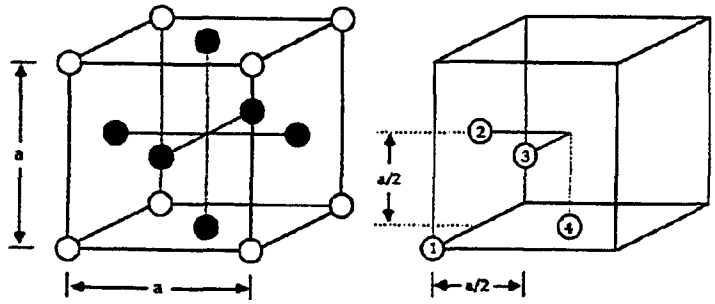


Figure 3.4: FCC lattice

This version of Verlet's algorithm depends on two previous time steps, and is thus not self-starting. Also note that the position does not explicitly depend on the velocity.

The above can be rewritten to obtain the more commonly used version of Verlet's method, the so-called "velocity Verlet" algorithm:

$$\vec{r}(t+\Delta t) = \vec{r}(t)+\dot{\vec{r}}(t)\Delta t+\ddot{\vec{r}}(t)\Delta t^2/2 \tag{3.17}$$

$$\dot{\vec{r}}(t+\Delta t) = \dot{\vec{r}}(t)+\left[\ddot{\vec{r}}(t)+\ddot{\vec{r}}(t+\Delta t)\right]\Delta t/2 \tag{3.18}$$

In order to conserve space, the velocity calculation is split into two parts, so that the acceleration is only stored for one time. The algorithm is shown in figure 3.5.

3.2.2 Gear's algorithm

Gear's algorithm belongs to the category of predictor-corrector methods. First the new values for the position and its derivatives are estimated. Then the forces are computed at the new locations, which allows for corrections to be computed by comparing the calculated with the predicted forces.

1. Initialize positions and velocities
2. Compute forces and energies
3. Loop:
   (a) Advance position using:

$$\vec{r}(t+\Delta t) = \vec{r}(t) - \dot{\vec{r}}(t)\Delta t + \ddot{\vec{r}}(t)\Delta t^2/2$$

   (b) Integrate velocity for half a time step:

$$\dot{\vec{r}}(t+\Delta t/2) = \dot{\vec{r}}(t) + \ddot{\vec{r}}(t)\Delta t/2$$

   (c) Compute forces and energies
   (d) Advance velocity for another half time step:

$$\dot{\vec{r}}(t+\Delta t) = \dot{\vec{r}}(t+\Delta t/2) + \ddot{\vec{r}}(t+\Delta t)\Delta t/2$$

Figure 3.5: Pseudocode of *velocity Verlet* algorithm.

The predictions are made using Taylor expansions [Hai97]:

$$\begin{aligned}
\vec{r}(t+\Delta t) &= \vec{r}(t) + \dot{\vec{r}}(t)\Delta t + \ddot{\vec{r}}(t)\frac{\Delta t^2}{2!} + \vec{r}^{(iii)}(t)\frac{\Delta t^3}{3!} + \vec{r}^{(iv)}(t)\frac{\Delta t^4}{4!} + \vec{r}^{(v)}(t)\frac{\Delta t^5}{5!} \\
\dot{\vec{r}}(t+\Delta t) &= \dot{\vec{r}}(t) + \ddot{\vec{r}}(t)\Delta t + \vec{r}^{(iii)}(t)\frac{\Delta t^2}{2!} + \vec{r}^{(iv)}(t)\frac{\Delta t^3}{3!} + \vec{r}^{(v)}(t)\frac{\Delta t^4}{4!} \\
\ddot{\vec{r}}(t+\Delta t) &= \ddot{\vec{r}}(t) + \vec{r}^{(iii)}(t)\Delta t + \vec{r}^{(iv)}(t)\frac{\Delta t^2}{2!} + \vec{r}^{(v)}(t)\frac{\Delta t^3}{3!} \\
\vec{r}^{(iii)}(t+\Delta t) &= \vec{r}^{(iii)}(t) + \vec{r}^{(iv)}(t)\Delta t + \vec{r}^{(v)}(t)\frac{\Delta t^2}{2!} \\
\vec{r}^{(iv)}(t+\Delta t) &= \vec{r}^{(iv)}(t) + \vec{r}^{(v)}(t)\Delta t \\
\vec{r}^{(v)}(t+\Delta t) &= \vec{r}^{(v)}(t)
\end{aligned} \tag{3.19}$$

Then the force is computed using these predicted values. Now the difference between the predicted value and computed value is determined:

$$\Delta\ddot{\vec{r}} = \ddot{\vec{r}}(t+\Delta t) - \ddot{\vec{r}}^{P}(t+\Delta t) \tag{3.20}$$

The superscript P denotes the predicted value. This now allows us to compute corrections terms, so that we get the following for the corrected values of the positions and their derivatives:

$$\begin{aligned}
\vec{r} &= \vec{r}^{P} + \alpha_0 \Delta\vec{R}2 \\
\dot{\vec{r}}\Delta t &= \dot{\vec{r}}^{P}\Delta t + \alpha_1 \Delta\vec{R}2 \\
\ddot{\vec{r}}\frac{\Delta t^2}{2!} &= \ddot{\vec{r}}^{P}\frac{\Delta t^2}{2!} + \alpha_2 \Delta\vec{R}2 \\
\vec{r}^{(iii)}\frac{\Delta t^3}{3!} &= \vec{r}^{(iii)P}\frac{\Delta t^3}{3!} + \alpha_3 \Delta\vec{R}2 \\
\vec{r}^{(iv)}\frac{\Delta t^4}{4!} &= \vec{r}^{(iv)P}\frac{\Delta t^4}{4!} + \alpha_4 \Delta\vec{R}2 \\
\vec{r}^{(v)}\frac{\Delta t^5}{5!} &= \vec{r}^{(v)P}\frac{\Delta t^5}{5!} + \alpha_5 \Delta\vec{R}2
\end{aligned} \tag{3.21}$$

where $\Delta\vec{R}2$ is defined as

$$\Delta\vec{R}2 \equiv \frac{\Delta\ddot{\vec{r}}\Delta t^2}{2!} \tag{3.22}$$

The values for α_i are given in table 3.2 [Hai97].

Figure 3.6 shows how Verlet's and Gear's methods compare in accuracy [Hai97]. Plotted are the root-mean-square (RMS) of the global error of the total energy as a function of time increments Δt. Gear's algorithm is clearly more accurate than Verlet's method. Typical time steps for our simulations are on the order of 0.5 fs, which in reduced units corresponds to 6.5×10^{-3}. At this step

α_i	$q=3$	$q=4$	$q=5$
α_0	$\frac{1}{6}$	$\frac{19}{120}$	$\frac{3}{16}$
α_1	$\frac{5}{6}$	$\frac{3}{4}$	$\frac{251}{360}$
α_2	1	1	1
α_3	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{11}{18}$
α_4	—	$\frac{1}{12}$	$\frac{1}{6}$
α_5	—	—	$\frac{1}{60}$

Table 3.2: Parameters for the correction terms in Gear's algorithm for predictions of various orders q.

size the error is about one order of magnitude less for Gear's 5th order method than for Verlet's.

3.2.3 Interaction computations

A significant part of the simulation time is spent on computing the forces acting the particles. Thus it is of interest to find ways to minimize the amount of interaction computations. As given by the potentials defined in sections 3.1.3 and 3.1.4, the net force acting on a particle depends on the location of all other particles in the system. In the simplest version (brute force), all particles are used to compute the forces, which for a two-body potential takes $O(N^2)$ time, where N is the number of particles in the system. If we note that the forces decrease with increasing distance between the particles and virtually go to zero after only a few σ, we can make a reasonable approximation of the net force by only considering the particles within a certain cutoff radius r_c. Two possible algorithms for this are cell-subdivision and neighbour-list, illustrated in figure 3.7.

In the neighbour-list algorithm, each atom carries a list of its neighbours. A particle is consid-

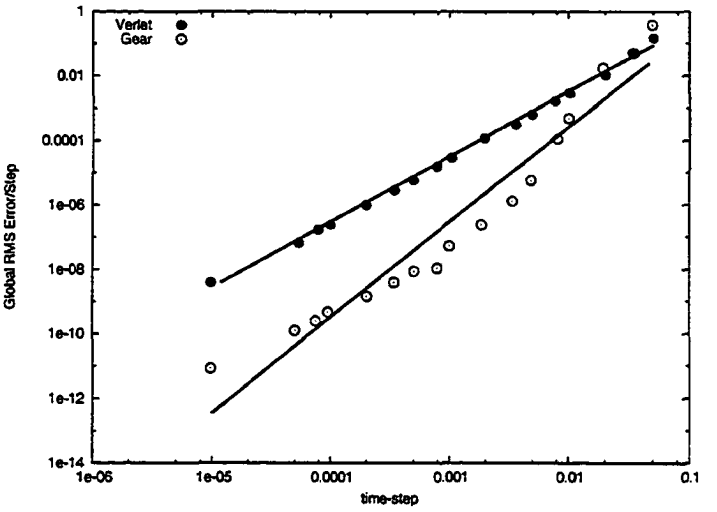


Figure 3.6: RMS of global error as a function of time step. Gear's algorithm (5th order) outperforms Verlet's algorithm in accuracy based on energy conservation. The lines are least-squares fit with slopes of 2.04 and 2.97 for Verlet's and Gear's method, respectively. Values are in reduced units. The time is given in multiples of $\tau = 1/(\sigma\sqrt{m/\varepsilon})$, and the energy is in terms of ε [Hai97]

ered a neighbour if it lies within a radius r_n, where of course $r_n > r_c$. This list of neighbouring atoms must be updated frequently, since particles move around and may change their neighbour status with respect to other particles. In order to decide when a neighbour list update is necessary, the maximum possible distance d_{max} travelled by any particle in the system is kept track of. When d_{max} exceeds $r_n - r_c$ then the neighbour lists need to be refreshed.

In cell-subdivision, the simulation volume is divided into a number of cells. Each of the particles is labeled according to which cell it is located in. If a particle moves from one cell to another, its label changes accordingly. In the force computations, only particles from the own cell and neighbouring cells are considered and included in the force computation only if they are within a cutoff radius. This is the algorithm employed in our simulation code.

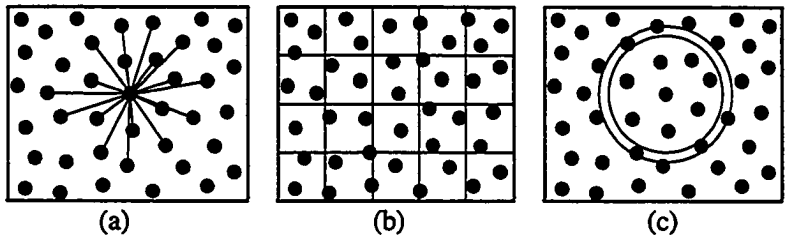


Figure 3.7: Illustrated are possible algorithms for the interaction computations: (a) all pairs, (b) cell subdivision, (c) neighbour list [Rap95].

In order for these two algorithms to be effective, the simulation volume must be several times larger than the interaction volume ($4\pi r_c^3/3$) specified by the cutoff radius r_c (for SW potential $r_c = a$ as given in table 3.1).

3.3 Simulation configuration

3.3.1 Periodic boundaries

Due to restricted computing power, a molecular dynamics simulation can only contain a limited number of particles. Our simulations typically contain several tens of thousand up to a few hundred thousand particles. While this may seem like a large number, it is still very small compared to the number of particles in a physical system (Avogardo's number is $N_A \approx 6 \times 10^{23}$). For a simulation with this few particles, a considerable fraction of them will be near the surface. The behaviour of the system would then be dominated by surface effects, which may not be desirable. This can be avoided by applying periodic boundary conditions, which eliminates the surfaces and makes the simulation volume virtually infinite. In order to obtain good statistics, one would still like to have as many particles as possible. A few hundred particles can be sufficient, depending on the problem and the

accuracy required in the study. For our problem, we simulate a small portion inside the laser focal spot region. Figure 3.8 illustrates the layout of the system when periodic boundary conditions are

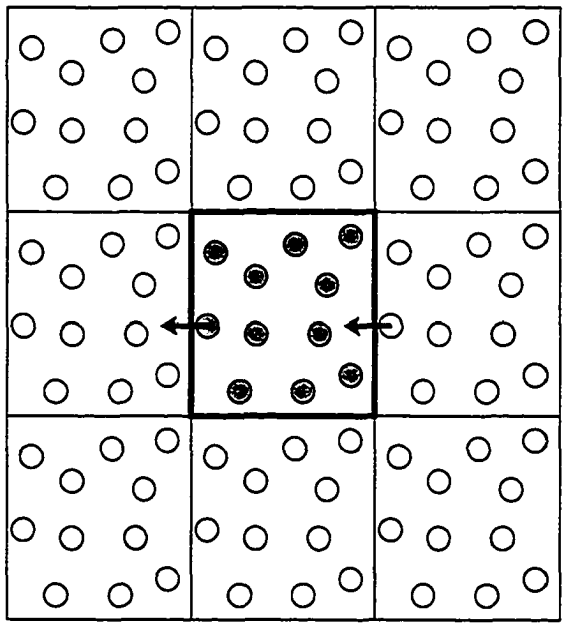


Figure 3.8: Periodic boundary conditions. Shown is a 2D representation of the system. If a particle leaves the simulation volume (cell), it automatically reenters the cell from the opposite side.

applied. Shown is a two-dimensional representation of the three-dimensional simulation. Whenever a particle leaves the simulation volume (cell) it automatically reenters it from the opposite side. At each step of the path integration the coordinates of each particle are checked to make sure it is still in bounds. If one is found to be out of bounds, then its coordinates must be adjusted.

3.4 Extraction of thermodynamic properties

3.4.1 Pressure

The pressure can be calculated in terms of the virial expression [Rap95]:

$$PV = Nk_BT + \frac{1}{D}\langle W \rangle \tag{3.23}$$

where D is the dimensionality, $\langle W \rangle$ is the virial and is given by:

$$\langle W \rangle = \left\langle \sum_{i=1}^{N} \vec{r}_i \cdot \vec{F}_i \right\rangle \tag{3.24}$$

In the case where the system's center of mass is not stationary, the virial needs to be adjusted as follows:

$$\langle W \rangle = \left\langle \sum_{i=1}^{N} \vec{r}_i \cdot \left(\vec{F}_i - \frac{\vec{F}_{CM}}{N} \right) \right\rangle \tag{3.25}$$

where $\vec{F}_{CM}$ is the net force acting on the system of particles.

Since in our simulations we use periodic boundary conditions, we cannot use eqn. 3.25 in the given form, but must express the virial in terms of relative positions:

$$\langle W \rangle = -\left\langle \sum_{i<j} \vec{r}_{ij} \cdot \vec{f}_{ij} \right\rangle \tag{3.26}$$

3.4.2 Heat capacity

The heat capacity describes how a system's temperature varies upon heating. It is defined as the the ratio of internal energy change (or external energy added to the system by heating) and resulting change in temperature:

$$C_X \equiv \left(\begin{array}{c} \text{heat capacity} \\ \\ \text{under condition } X \end{array} \right) = \left(\frac{q}{\Delta T}\right)_X = \left(\frac{\Delta E}{\Delta T}\right)_X = \left(\frac{\partial E}{\partial T}\right)_X \tag{3.27}$$

Often X denotes constant volume or constant pressure.

We can estimate the isometric heat capacity for a crystalline solid using the equipartition theorem: With each atom there are three degrees of freedom associated for both the kinetic and potential energy:

$$\langle E_{\text{kin}}(t)\rangle = \langle E_{\text{pot}}(t)\rangle = \frac{3}{2} N k_B T \tag{3.28}$$

Where $\langle E_{\text{pot}}(t)\rangle$ and $\langle E_{\text{kin}}(t)\rangle$ are the average potential and kinetic energy of the solid, respectively, T is the temperature, and N is the number of particles. Thus the heat capacity of a solid is

$$C_V = 3Nk_B \tag{3.29}$$

Figure 3.9 shows the average energy per Si atom (potential + kinetic) versus the temperature of the system. The data points in the figure were obtained from molecular-dynamics runs in which the total energy was monotonically increased. The energy was added by applying small random forces to the particles using the Langevin damping technique (see sec. 4.1) for 500 time steps (1

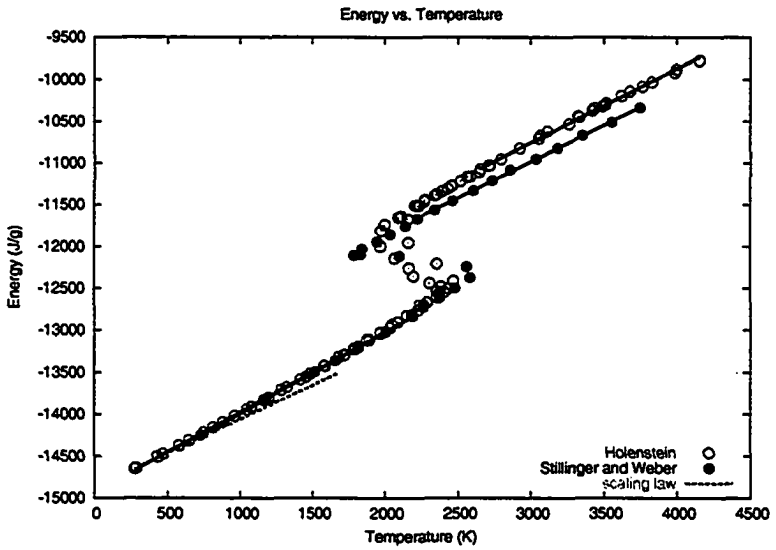


Figure 3.9: Average energy density vs. temperature. The system of 1000 atoms was systematically heated from case to case (with equilibration). Our measurements are compared to Stillinger and Weber's data [SW85] and to the empirical scaling law given in eqn. 3.30 (integrated starting from our first data point) [NC92].

fs each), followed by an equilibration period of 500 time steps. The energy and temperature was then averaged over 1000 steps. For these simulations we used the same parameters as Stillinger and Weber (see table 3.1. Two independent data sets were produced. A difference between the two sets is visible in the transition region (1900-2500 Kelvin), where a phase transformation from solid to liquid takes place. In one of the two simulations the system remains solid slightly longer than in the other simulation. The solid circles are the results obtained by Stillinger and Weber (fig. 3 of [SW85]). Our data agrees with Stillinger and Weber's for the solid branch, but deviates after the phase transition. However, for both phases the slopes are close to the same for both models. The solid lines are linear fits (least squares) for the solid and liquid branches of the data sets. The heat capacities obtained from the slopes of these lines are given in table 3.3. They are in good agreement

	Heat Capacity [J/g·K]	
	solid	liquid
Stillinger and Weber	0.968	0.872
Holenstein	0.951	0.885

Table 3.3: Isometric heat capacities determined from energy-temperature graph (fig. 3.9)

with Stillinger's data as obtained from the graph. The dashed line represents an empirical scaling law for the heat capacity as a function of temperature. It is the integral of eqn 3.30 [NC92] with the starting point set to match the first data point of our data set.

$$c_V = \left(1.978\frac{\mathrm{J}}{\mathrm{cm^3K}}\right) + \left(2.54\times10^{-4}\frac{\mathrm{J}}{\mathrm{cm^3K^2}}\right)T - \left(3.68\times10^{4}\frac{\mathrm{JK}}{\mathrm{cm^3}}\right)T^{-2}, \quad 300\mathrm{K}<T<1685\mathrm{K} \tag{3.30}$$

Figure 3.10 shows the heat capacity with respect to temperature. This was obtained from the derivative of the solid branch in figure 3.9.

Stillinger and Weber quote values for the residual heat capacity (i.e. the heat capacity with the contribution from kinetic energy removed[1]) in reduced units [SW85]. The reduced residual heat capacity is defined as [2]

$$C_V^{*R} \equiv C_V^* - \frac{3}{2} \equiv \frac{C_V^R}{Nk_B} \tag{3.31}$$

where $N = 216$ is the number of atoms in the system and k_B is Boltzmann's constant.

Noya et al. [NHR96] have also computed the heat capacity, among other properties, of silicon

[1] Expressing the heat capacity in the form of residual heat capacity emphasizes the contribution from the particle interactions.

[2] The paper [SW85] defines the heat capacity as $C_V^{*R} \equiv \frac{C_V^R}{Nk_B\varepsilon}$. However, the units don't work out and the ε appears to be a typographical error.

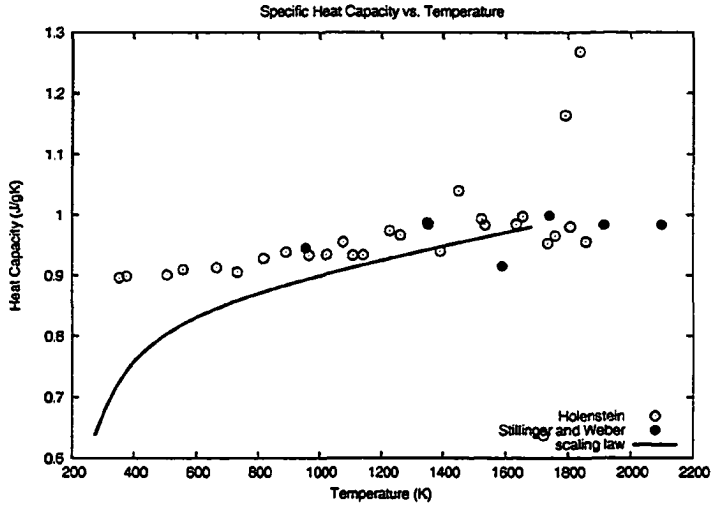


Figure 3.10: Isometric heat capacity versus temperature. Our measurements are compared to Stillinger and Weber's data [SW85] and to the empirical scaling law given in eqn. 3.30 [NC92]. The noise in the heat capacity increases as the temperature gets closer to the melting point and the system approaches a phase transition.

using the SW potential. They use path-integral Monte Carlo simulations in an isothermal-isobaric ensemble. The reported values for the isobaric heat capacity c_p are in reasonable agreement with experimental values, as shown in figure 3.11

3.4.3 Thermal conductivity

In order to determine the thermal conductivity of our system, we use the following approach: The silicon bulk is coupled to a heat bath at the top and bottom, with periodic boundaries in the horizontal (x-y) directions. A sinusoidal temperature profile is used as the initial condition:

$$T(z,0) = T_0 + \Delta T \sin\left(\frac{\pi z}{L}\right) \tag{3.32}$$

	C_V^{*R} [SW85]	c_V $\left(\frac{J}{g \cdot K}\right)$
low temperature crystal	1.5	0.888
crystal at melting temperature	2.0	1.036
liquid at melting temperature	1.6	0.918

Table 3.4: Specific heat capacities (c_V) for silicon obtained from reduced residual heat capacities (C_V^{*R}) given by Stillinger and Weber [SW85]. The low temperature crystal value is obtained from the equipartition theorem. (Equation 3.31 was used to compute the real values)

where L is the height (or length) of the system and ΔT is the initial temperature difference between the heat bath and the center of the volume (i.e. $\Delta T = T(L/2,0) - T(0,0)$). Solving the heat flow equation (eqn. 2.3) for these initial conditions and assuming that the thermal conductivity (κ) and heat capacity (c_V) are (approximately) constant over the temperature range $[T_0, T_0 + \Delta T]$, we get (see appendix B):

$$T(x,t) = T_0 + \Delta T \sin\left(\frac{\pi z}{L}\right) \exp\left(-\frac{\pi^2 \kappa t}{\rho c_V L^2}\right) \tag{3.33}$$

where ρ is the density. We see that the peak temperature (at the center of the simulation volume) shows an exponential decrease over time and asymptotically approaches T_0. By measuring the decay lifetime and heat capacity (section 3.4.2), we can calculate the thermal conductivity. Figure 3.12a shows a sinusoidal least-squares fit to the temperature profile. The boundary condition was set to $T_0 = 300\,\mathrm{K}$ and $\Delta T = 100\,\mathrm{K}$. There is some significant noise in the data, which is due to the limited number of particles (per cross-sectional area) used in the simulation. The peak values (at $z = L/2$) of such fitted curves is plotted in figure 3.12b and fitted to an exponential. Using a value of $c_V = 0.90\,\mathrm{J/(gK)}$ for the heat capacity, we obtain $\kappa = 0.139\,\mathrm{W/(cmK)}$ from the exponent of the fitted exponential. This value is lower than that reported for crystalline silicon ($1.5\,\mathrm{W/(cmK)}$) by

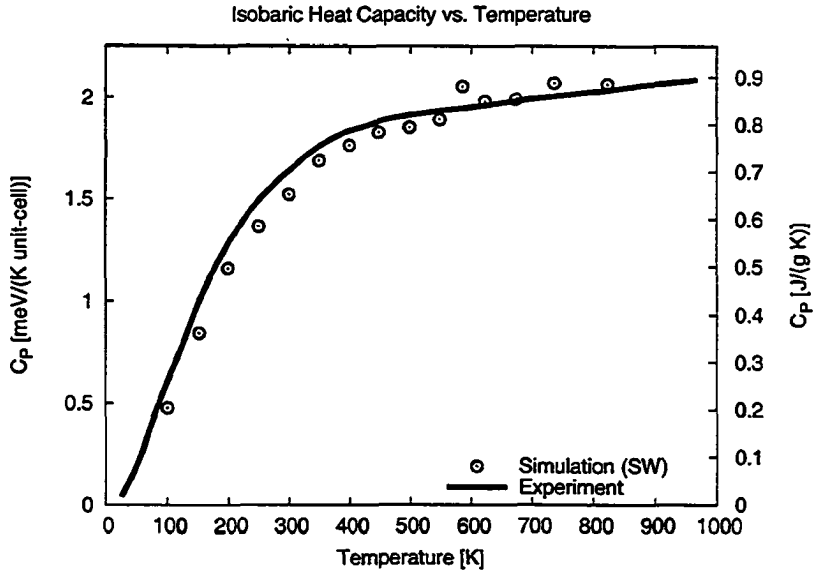


Figure 3.11: Isobaric heat capacity at 1 atm. pressure as given by Noya et al. [NHR96]. The results from simulations using the SW potential (open circles) are compared to experimental results (solid line).

about a factor of ten, but higher than amorphous silicon (0.018 W/(cmK)) by an order of magnitude [Bäu00]. The system is far below the melting point, and was initialized as a crystal, thus the thermal conductivity should be ~1.5 W/(cmK). Clearly the SW potential strongly underestimates the thermal conductivity. This, at least at low temperatures, has implications for the energy transport in from the laser heated region into the silicon bulk. While this may not be a major factor for ultra-short pulses, it likely affects the results for longer pulses, where heat transport plays a stronger role. Thus we would expect the threshold fluence for melting or ablation of c-Si to be lower than in experiment, and higher for a-Si. However, the author has not found any reports on this.

The thermal conductivity can also be obtained from equilibrium simulations. One such approach

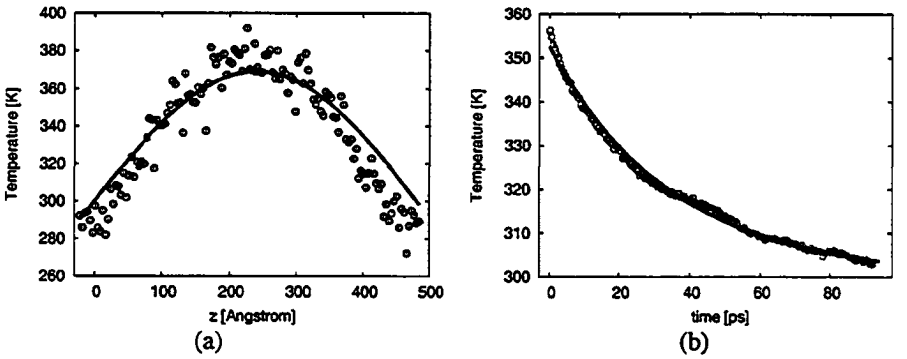


Figure 3.12: (a) Temperature profile after 1.5 ps with sinusoidal fit. (b) Temperature of sine fit at $z = T/2$ as a function of time and fitted to exponential. The exponent is -2.836×10^{-5} which corresponds to a conductivity of $\kappa = 0.139\,\mathrm{W/(cmK)}$ ($c_V = 0.90\,\mathrm{J/(gK)}$, $\rho = 2.32\,\mathrm{g/cm^3}$).

is based on the Green-Kubo formulation [Kub57] and has been utilized in several studies [Zwa65, PB94, VC00]. The spectral thermal conductivity is given as [VC00]:

$$\kappa(\vec{k},\omega) = \frac{V}{3k_B T_0^2} \int_0^\infty \left\langle \vec{q}_0(\vec{k},0) \cdot \vec{q}_0(\vec{k},t) \right\rangle e^{i\omega t} dt \tag{3.34}$$

where $\vec{q}_0$ is the equilibrium heat flux, and $\vec{k}$ and ω are the wave vector and frequency of the external (thermal) perturbation exerted on the system. Volz and Chen have computed the conductivity for the SW potential using the spectral Green-Kubo formalism in combination with their own correction terms to eliminate size effects and artifacts due to periodic boundary conditions (PBC). In MD simulations with PBC only phonons with a wavelength shorter than the simulation volume size are are allowed to exist, cutting off low-frequency phonons. Further, the PBC introduce an artificial autocorrelation, which is not present in real systems. This results in a low value for the heat conductivity. After correcting for this, Volz and Chen have found their values to agree well with

isotopically enriched ^{28}Si, but are found to be higher than the heat conductivity measured in natural silicon. Their results are summarized in figure 3.13 [VC00]. Our result is in agreement with theirs

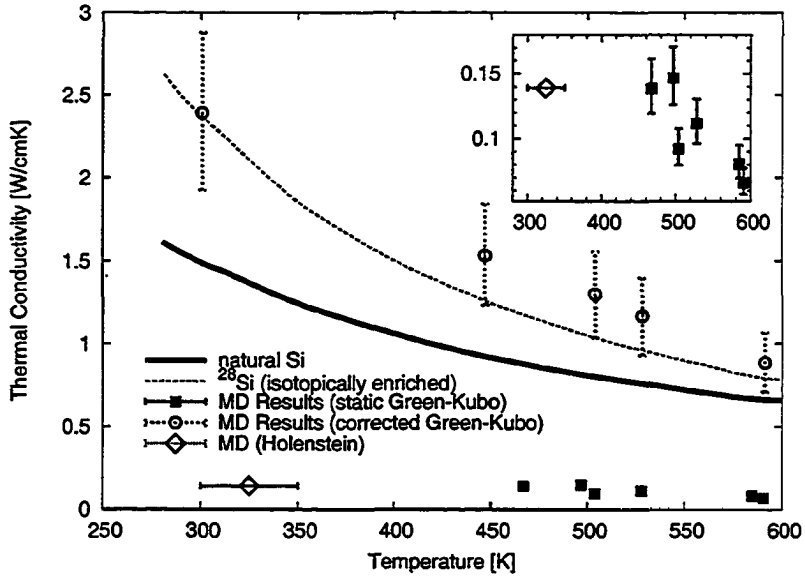


Figure 3.13: Thermal conductivity from MD simulations using Green-Kubo auto-correlation by Volz et al. and experimental data from natural and isotopically enriched silicon [VC00].

for the static Green-Kubo (non-corrected) formulation.

3.5 Optimization: Lookup table for SW potential and force

In an effort to speed up the computation, a lookup table has been implemented. The SW potential and force are precomputed for radii up to the SW cutoff and for bond angles $0 - \pi$. For the two-body interaction, the lookup table was indexed by the distance between the particle and its interacting neighbour. The three-body interaction had three coordinate indices (r_{ij}, r_{ik}, $\cos\alpha$) and returned five

values: the potential energy ϕ and the forces acting on the neighbouring particles in terms of the vector components along $\vec{r}_{ij}$ and $\vec{r}_{ik}$, i.e. $\vec{f}_j = f_j^{(j)}\hat{r}_{ij} + f_j^{(k)}\hat{r}_{ik}$ and similar for $\vec{f}_k$.

$$\text{lookup}(r_{ij}, r_{ik}, \cos\theta_{jik}) \rightarrow \{f_j^{(j)}, f_j^{(k)}, f_k^{(j)}, f_k^{(k)}, \phi\} \tag{3.35}$$

The force on particle i (see fig. 3.14) is simply $\vec{f}_i = -(\vec{f}_j + \vec{f}_k)$

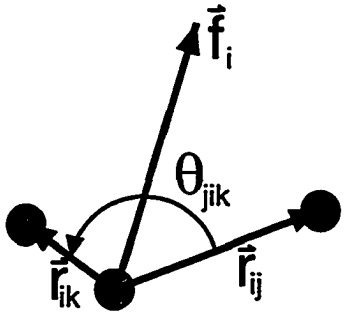


Figure 3.14: Geometry of interacting particles using SW potential.

3.5.1 Nearest neighbour interpolation

In nearest neighbour interpolation, the data values at the nearest grid point are returned. This means that the force and potential are effectively represented as step functions. In order to compensate for this, a higher resolution in the lookup table is required than for higher order interpolation. However, only one lookup and less computation is needed, thus making this method much faster.

3.5.2 Linear interpolation

For the linear interpolation approach the data values need to be looked up for eight different coordinates. Let $c_{xxx} = (\alpha, r_1, r_2)$ be the coordinate for which the data values are to be interpolated. The three symbols in the subscript represent the coordinates α, r_1, and r_2, respectively. An x denotes interpolated coordinate, a 0 and 1 denote grid points (in the lookup table) just below and above the interpolated coordinate, respectively, as illustrated in figure 3.15. The data values at coordinate c_{xxx},

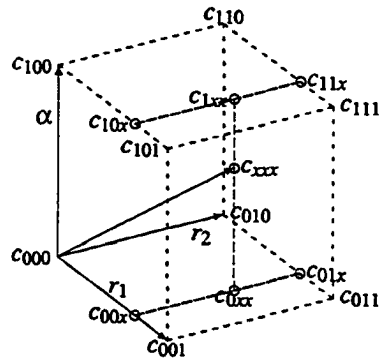


Figure 3.15: Coordinates for linear interpolation.

represented as d_{xxx}, are obtained as follows:

$$\begin{aligned} d_{00x} &= d_{000} + (d_{001} - d_{000})q_{r_2} \\ d_{01x} &= d_{010} + (d_{011} - d_{010})q_{r_2} \\ d_{0xx} &= d_{00x} + (d_{01x} - d_{00x})q_{r_1} \\ d_{10x} &= d_{100} + (d_{101} - d_{100})q_{r_2} \\ d_{11x} &= d_{110} + (d_{111} - d_{110})q_{r_2} \\ d_{1xx} &= d_{10x} + (d_{11x} - d_{10x})q_{r_1} \\ d_{xxx} &= d_{0xx} + (d_{1xx} - d_{0xx})q_{\alpha} \end{aligned} \tag{3.36}$$

where q_{r_1}, q_{r_2}, and q_α are given by

$$\begin{aligned} q_{r_1} &= \frac{r_1 - r_1^{[c_{000}]}}{r_1^{[c_{001}]} - r_1^{[c_{000}]}} \\ q_{r_2} &= \frac{r_2 - r_2^{[c_{000}]}}{r_2^{[c_{010}]} - r_2^{[c_{000}]}} \\ q_{\alpha} &= \frac{\alpha - \alpha^{[c_{000}]}}{\alpha^{[c_{100}]} - r_1^{[c_{000}]}} \end{aligned} \tag{3.37}$$

The superscripts denote the value of the coordinate at the given point, e.g. $r_1^{[c_{001}]}$ is the value of coordinate r_1 at point c_{001}.

3.5.3 Results

Comparing the performance of the lookup table with the dynamical force calculation revealed that the lookup table is at least six to seven times slower for both the linear and nearest neighbour interpolation. The limiting factor is the memory access time. For sufficient accuracy, the size of

the lookup table for the 3-body potential was on the order of 140 MB (double precision) with a resolution of 200 data points for the radial indices and 180 point for the angular index. Two tables were used, the second one for the case of broken bonds. Given the disappointing performance, the lookup tables were not used for the simulations.

3.6 Optimization: Parallelization

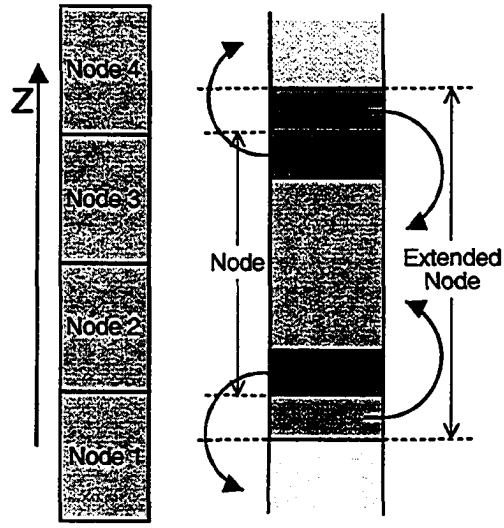


Figure 3.16: Layout of nodes and communication in parallel simulation.

In order to speed up the computation, the code was parallelized using MPI (Message Passing Interface). The simulation volume was split up along the z-direction, as shown in figure 3.16. At each time step the updated positions of the particles near the node boundaries are passed to the neighbouring node. Since the interaction range is not the same for all particle species (i.e. the

Coulomb potential extends farther than the SW potential) the the size of the extended node differs depending on particle type. The overlapping region is taken to be a multiple of the cell size used for the interaction computation (see section 3.2.3). The parallel code scales well with number of

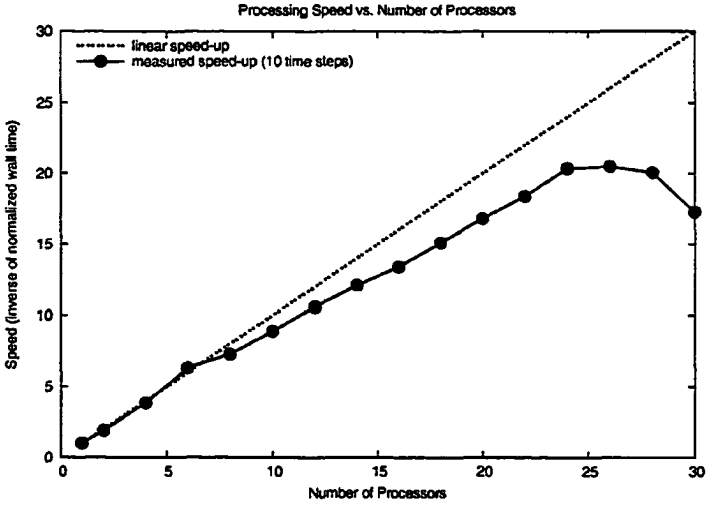


Figure 3.17: Speed-up of simulation with number of processors. The simulation was run on an SGI Origin 2400.

processors. Figure 3.17 shows the speed-up as a function of processors used. The measured wall time[3] is over 10 time steps. It does not include the time spent initializing and writing results to file, which are negligible over long runs. The simulations involved 172800 atoms and the bulk had a height to width ratio of 12.5. Using a single processor the wall time for one time step was about 800 seconds.

[3]Wall time: The real running time of a program, as measured by a clock on the wall, as opposed to the number of ticks or CPU time required to execute it.

Chapter 4

Heat Bath and Coupling between HF and MD model

The heat dissipation in and out of the MD system is implemented using Langevin dynamics, as described in the following section. A similar approach is taken for thermally coupling the MD model with the heat flow model when it is added at the bottom of the simulation. This is described in section 4.2.

4.1 Langevin damping

The equation of motion for the atoms in the boundary region is governed by the Langevin equation [AD76, HIM95]. The random force and the friction coefficient are given by the fluctuation-

dissipation theorem [Kub66]:

$$m_k\ddot{\vec{r}}_k = \vec{F}_k(\vec{r}_1,...,\vec{r}_N,) - m_k\gamma_k\dot{\vec{r}}_k + \vec{R}_k \tag{4.1}$$

where

$$\gamma_k = \frac{\pi}{6}\omega_D \tag{4.2}$$

and $\vec{R}_k$ are random white noise forces given by:

$$\vec{R}_k = \sqrt{2\gamma_k k_B T m_k}\,\vec{\eta}_k \tag{4.3}$$

$\vec{\eta}_k$ are vectors of Gaussian random numbers centered at zero with

$$(\langle\vec{\eta}_m(t)\vec{\eta}_k(t')^T\rangle)_{ij} = \delta(t-t')\delta_{ij}\delta_{mk} \tag{4.4}$$

Note that $\vec{\eta}_k$ has units of $s^{-1/2}$, since the definition of the Dirac delta function requires that the product $\delta(t)dt$ is dimensionless. The Debye frequency is given by $\omega_D = k_B\theta_D/\hbar$. The Debye temperature θ_D can be obtained experimentally and for silicon is found to be $\theta_D = 645$K. In the MD simulation this is implemented as:

$$\Delta\ddot{\vec{r}}_k = -\gamma_k\dot{\vec{r}}_k \tag{4.5}$$

$$\Delta\dot{\vec{r}}_k = \sqrt{\frac{2\gamma_k k_B T\Delta t}{m_k}}\vec{\xi}_k \tag{4.6}$$

The vector components ξ_i are Gaussian random numbers. T is the environment temperature. When using the heat flow (HF) model, the temperature T is obtained from the HF system.

4.2 HF boundary condition

The energy transfer from the MD system to the HF system in the overlapping region is obtained by considering the average kinetic (thermal) energy gained due to thermal fluctuations in the environment, which in this case is the MD system. Let $\vec{v}_R$ be the velocity gain due to random forces $\vec{R}$ acting on the particle over a time interval Δt. Given that the particle had an initial velocity of $\vec{v}$, the energy gain after during Δt is then

$$\langle \Delta E \rangle = \frac{1}{2} m \left(\left\langle |\vec{v} + \vec{v}_R|^2 \right\rangle - |\vec{v}|^2 \right) \tag{4.7}$$

Since $|\vec{v}_R| \ll |\vec{v}|$ we may assume that

$$\left\langle |\vec{v} + \vec{v}_R|^2 \right\rangle - |\vec{v}|^2 \approx \left\langle |\vec{v}_R|^2 \right\rangle \tag{4.8}$$

Since the vector components of $\vec{R}$ (and $\vec{v}_R$) of are not correlated, we find that

$$\left\langle |\vec{v}_R|^2 \right\rangle = 3 \left\langle |v_{Ri}^2| \right\rangle \tag{4.9}$$

Averaging the square of the random force over one timestep:

$$\langle R^2 \rangle = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} R(t)^2 \, dt = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} 2\gamma_k k_B T_{\mathrm{MD}} m \delta t \, dt = \frac{2\gamma_k k_B T_{\mathrm{MD}} m}{\Delta t} \tag{4.10}$$

Now integrating the random force over time Δt gives

$$\langle v_{Ri}^2 \rangle = \left\langle \left(\frac{R \Delta t}{m} \right)^2 \right\rangle = \frac{\Delta t^2}{m^2} \langle R^2 \rangle = \frac{2}{m_k} \gamma_k k_B T_{\mathrm{MD}} \Delta t \tag{4.11}$$

So the energy gain due to heat flowing from MD to HF system is:

$$\langle \Delta E_{\mathrm{in}} \rangle = \frac{1}{2} m \langle |\vec{v}_R|^2 \rangle = 3\gamma_k k_B T_{\mathrm{MD}} \Delta t \tag{4.12}$$

Similarly for the heat transfer out of the HF system, thus the net energy gain (per atom) in the HF system is

$$\langle \Delta E \rangle = 3\gamma_k k_B \left(T_{\mathrm{MD}} - T_{\mathrm{HF}} \right) \Delta t \tag{4.13}$$

4.2.1 Derivation of Langevin damping

Let $W(u_0, t_0; u, t)$ be the transition probability for a particle having velocity u_0 at time t_0 to a velocity u at time t. The transition probability is a fundamental solution of the Fokker-Plank equation [Kub66]:

$$\frac{\partial}{\partial t} W = \frac{\partial}{\partial u} \left(D_u \frac{\partial}{\partial u} + \gamma_k u \right) W \tag{4.14}$$

$$W(u_0, t_0; u, t_0) = \delta(u - u_0) \tag{4.15}$$

where D_u is the diffusion constant in the velocity space and is given by the random force:

$$D_u = \frac{1}{m^2}\int_0^\infty \langle R(t_0)R(t_0+t)\rangle\, dt \tag{4.16}$$

At equilibrium, the velocity distribution must coincide with the Maxwellian distribution, i.e.

$$\lim_{t\to\infty} W(u_0,t_0;u,t) = Ce^{-\frac{1}{2}\frac{mu^2}{kT}} \tag{4.17}$$

As t goes to infinity, the left hand side of equation 4.14 goes to zero and we get:

$$0 = \lim_{t\to\infty}\left\{\frac{\partial}{\partial u}\left(D_u\frac{\partial}{\partial u}+\gamma_k u\right)W\right\} \tag{4.18}$$

$$0 = \frac{\partial}{\partial u}\left(D_u\frac{\partial}{\partial u}+\gamma_k u\right)Ce^{-\frac{1}{2}\frac{mu^2}{kT}} \tag{4.19}$$

$$0 = \frac{\partial}{\partial u}\left(D_u\frac{\partial}{\partial u}e^{-\frac{1}{2}\frac{mu^2}{kT}}+\gamma_k u e^{-\frac{1}{2}\frac{mu^2}{kT}}\right) \tag{4.20}$$

$$0 = \left(\gamma_k - D_u\frac{m}{kT}\right)\frac{\partial}{\partial u}\left(ue^{-\frac{1}{2}\frac{mu^2}{kT}}\right) \tag{4.21}$$

$$D_u = \frac{\gamma_k}{m}kT \tag{4.22}$$

4.3 Coupling

In order to connect the molecular dynamics simulation with the heat diffusion model, we use the following scheme, as illustrated in figure 4.1: A section at the bottom of the MD simulation overlaps with the top portion of the heat flow model. The particles at the bottom of the MD simulation are damped using Langevin dynamics (see section 4.1) and the temperature obtained from the heat

flow model. The average temperature of the particles contained in the damped region of the MD simulation (overlapping with HF model) is used to couple energy into the heat flow model. Refer to section 4.1 for details.

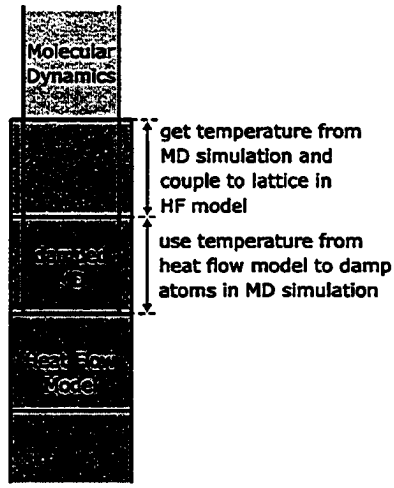


Figure 4.1: Molecular Dynamics-Heat Flow hybrid model: layout of interfacing.

4.4 Testing

In order to test the coupling between the HF and MD system, a simulation was done in which the surface atoms were continually heated by "damping" them to 1200 Kelvin. The evolution of the temperature profile is shown in figure 4.2. The heat is conducted into the HF system. A small "kink" in the isotherms is visible, which is due to a small mismatch in the heat capacity and conductivity. Figures 4.3 and 4.4 compare the temperature evolution upon absorption of a laser pulse with and without having a heat flow model connected to the bottom of the MD simulation. There is a

clear improvement in the temperature profile when the HF model is included. Overall the coupling performs well and is adequate.

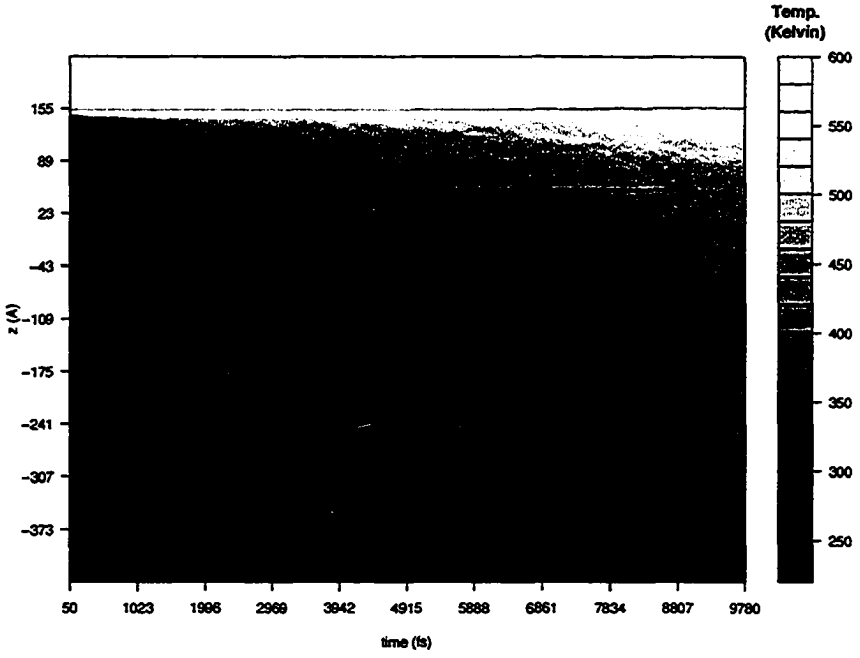
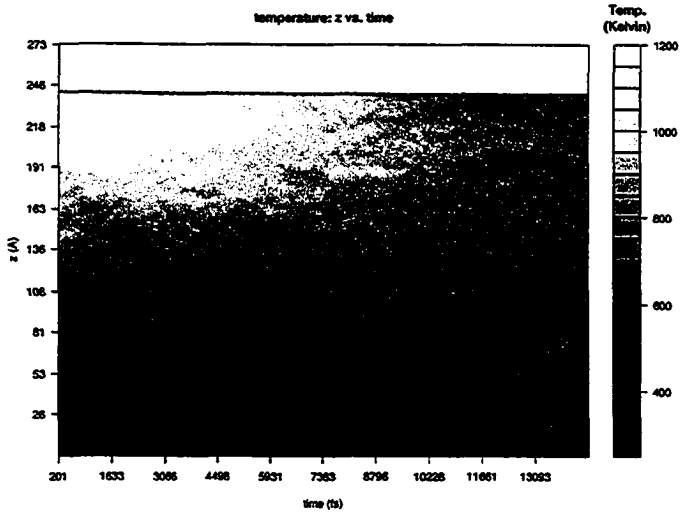


Figure 4.2: Evolution of temperature profile under continuous heating of the surface. The horizontal black line indicates the boundary between the MD system (above) and the HF system (below).

(a)



(b)

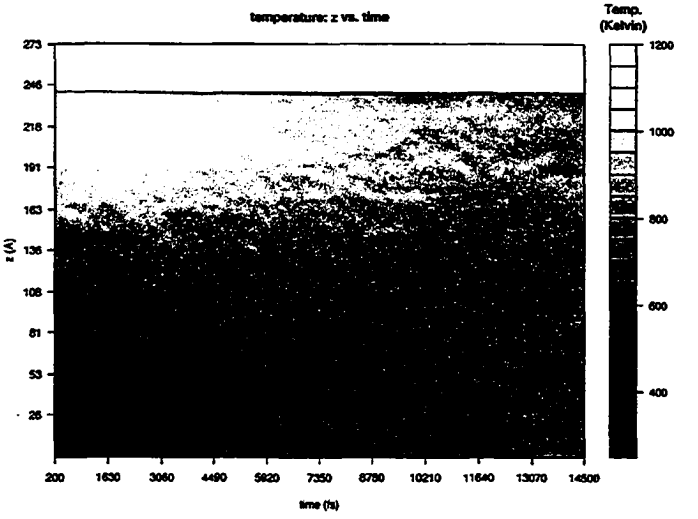


Figure 4.3: Evolution of temperature profile for (a) without and (b) with a heat flow model connected at the bottom of the MD system.

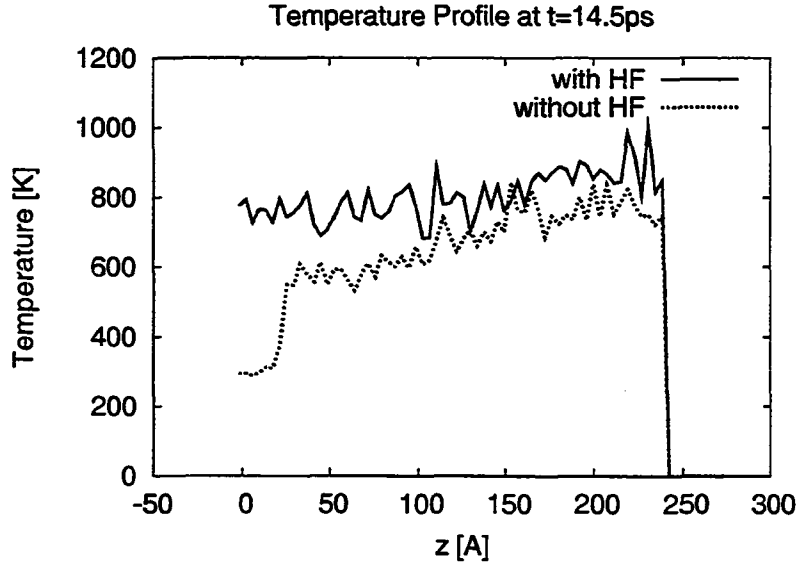


Figure 4.4: Temperature profile of figure at $t = 14.5$ ps.

Chapter 5

Laser Absorption

5.1 Plane wave propagation

The electric field of a wave propagating in a uniform, non-absorbing medium is given by

$$\vec{E}(z,t) = \vec{E}_0 e^{i(2\pi z/\lambda - \omega t)} \tag{5.1}$$

where t is the time and z is the spatial coordinate along the wave's direction of propagation. The wavelength depends on the index of refraction n:

$$\lambda = \frac{c}{n}\frac{1}{\nu} = \frac{c}{n}\frac{2\pi}{\omega} \tag{5.2}$$

In an absorbent medium, the index of refraction has a complex component:

$$n = n_1 + in_2 \tag{5.3}$$

Thus the wave equation can be written as:

$$\vec{E}(z,t) = \vec{E}_0 e^{i\left(\omega\frac{n_1}{c}zi-\omega t\right)} e^{-\omega\frac{n_2}{c}z} \tag{5.4}$$

The second exponential indicates an exponential decay in the electric field, and thus in the intensity:

$$I \propto e^{-2\omega\frac{n_2}{c}z} = e^{-\alpha z} \tag{5.5}$$

5.2 Absorption coefficient

The absorption coefficient α is given by [vAB95]

$$\alpha = -\frac{1}{I}\frac{dI}{dz} = 2\omega\frac{n_2}{c} \tag{5.6}$$

We can generally express the absorption coefficient as follows [Tsu02]:

$$\alpha = \alpha_0 + \sigma N_i + \alpha_D(N_t) + \alpha^{NL} \tag{5.7}$$

α_0 is the linear absorption coefficient and depends on the microstructure of the material (e.g. crystalline, amorphous, etc.), the wavelength, and the temperature. For metals and insulators α_0 is not

very sensitive to temperature. For semiconductors, α_0 is the sum of the interband absorption coefficient α_f and the free carrier absorption coefficient $\alpha_c = \sigma_a N_c$, where N_c is the free carrier number density and σ_a is the absorption cross-section. The temperature dependence arises mostly from the carrier density $N_c(T)$.

The second term in equation 5.7 is the absorption due to impurities or dopants of number density N_i with an absorption cross-section σ.

$\alpha_D(N_l)$ expresses the change in absorption resulting from radiation-induced defects, i.e. incubation effects, and depends on the laser intensity and number of laser pulses N_l.

Finally, the last term in eqn. 5.7 reflects multiphoton absorption processes. α^{NL} depends on the photon energy and laser intensity. The linear and two-photon absorption coefficients at $\approx$700 nm for Si are 3500 cm^{-1} and 55 cm/GW [HGC98, STBv95].

5.2.1 Implementation

As the laser energy is absorbed, the material will expand and thus the density and surface position do not remain constant. It is therefore not feasible to simply add the laser energy according to equation 5.5 (especially if one was to consider multiple laser pulses). Instead, we use a probabilistic approach: at each time step, the number of photons entering the system is computed. These photons then traverse the simulation volume, starting at the top and moving down in small steps (dz). The probability of linear and two photon absorption is calculated according to the local intensity. The number of photons absorbed by each of the atoms in that interval (layer) is determined and subtracted from the remaining number of photons. The step size is equal to the size of the

cells used for the interaction calculation, as outlined in section 3.2.3. The local intensity is simply $I(z) = N_p(z)\varepsilon_p/A_\phi$, where $N_p(z)$ is the number of photons (per time step Δt), ε_p is the energy of a single photon, and A_ϕ is the cross-sectional area. The absorption probabilities for the linear (P_1) and two photon (P_2) absorption per atom per photon is:

$$P_1 = 1 - \exp\left(-\frac{\alpha_0 m_{Si}}{\rho_0 A_\phi}\right) \tag{5.8}$$

$$P_2 = 1 - \exp\left(-\frac{\beta m_{Si}}{\rho_0 A_\phi}\frac{N_p(z)\varepsilon_p}{A_\phi}\right) \tag{5.9}$$

The derivation for this is as follows: Let $T = T_\alpha$ be the probability of transmission, i.e. the probability of a photon not being absorbed by an atom, for absorption coefficient α. Then the probability $P(z)$ of a given photon reaching depth z is

$$P(z) = T^{N_A(z)} \tag{5.10}$$

where $N_A(z) = \rho_0 A_\phi z/m_{Si}$ is the number of atoms encountered up to that depth. The total number of photons reaching depth z is $N(z) = N_0 P(z)$ and relating this to Beer's law:

$$N(z) = N_0 T^{N_A(z)} = N_0 e^{-\alpha z} \tag{5.11}$$

We can now solve for the transmission probability T and obtain:

$$T = T_\alpha = \exp\left(-\frac{\alpha z}{N_A(z)}\right) = \exp\left(-\frac{\alpha m_{Si}}{\rho_0 A_\phi}\right) \tag{5.12}$$

And the absorption probabilities are simply $P_1 = 1 - T_{\alpha_0}$ and $P_2 = 1 - T_{\beta I(z)}$, as given above.

5.3 Excitation and ionization

In a semiconductor, the absorption of a photon with energy larger than the band gap (1.12 eV for Si at 300 K) can occur by three possible mechanisms [LLM00b]:

- single photon absorption by a valence electron, followed by an interband transition promoting it to the conduction band,
- multi-photon absorption by a valance electron, followed by an interband transition to the conduction band and possible emission of a photon.
- absorption by conduction band electron (inverse bremsstrahlung)

Free carrier absorption has been shown to be negligible for photons at energies well above the band gap. For our simulations, only single and 2-photon absorption are considered. This is implemented as follows: Upon absorption of a photon, the excitation level of the atom is incremented (electron is promoted to the conduction band, leaving a hole behind). The generated electron-hole plasma affects the potential, as suggested by Stampfli [SB94], causing an instability in the lattice. This is implemented by randomly breaking a number of bonds matching the excitation level [HGC97, HGC98]. Since Si has four valence electrons, an atom at 4th excitation is considered unbound. A broken bond was implemented by removing the attractive part of the 2-body potential and discarding

the 3-body part of the SW potential.

$$\phi_i = \begin{cases} \text{original SW potential,} & \xi \geq \text{exitation level} \\ \text{modified SW potential,} & \text{otherwise} \end{cases} \tag{5.13}$$

where ϕ_i is the potential energy of particle i and ξ is a random number ($0 \leq \xi < 4$, uniform distribution).

If the total energy absorbed by an atom exceeds the work function (4.85 eV for Si) it becomes ionized, i.e. at the 4th excitation for $\lambda_L = 800\,\text{nm}$. The particle is then marked accordingly and its excitation level reset to zero. An electron is added at a distance of $1.17\,\text{Å}$ from the ejecting atom in a random direction initialized to be moving away from the ion. In addition to the SW potential the Coulomb potential is added.

It has been shown that avalanche ionisation will also play an important role in the heating and ablation of silicon surfaces [PVH+98]. It is expected that this will be important for longer pulse durations of several hundred femtoseconds leading to increased heating and absorption as compared with the current model simulation presented here. Niemz has presented a model to estimate the threshold for optical breakdown, taking into account avalanche ionization in combination with electron-ion recombination and electron diffusion [Nie95]. The threshold energy was found to have a square-root dependence on the pulse length in the picosecond and nanosecond range, where thermal diffusion dominates. Avalanche ionization is not included in the simulations presented in this thesis and is investigated in a future study.

5.4 Relaxation processes

There are several mechanisms that allow the system to restore equilibrium: carrier-phonon scattering, carrier diffusion, and Auger recombination. As a consequence of the Beer-Lambert law there will be a carrier density gradient, resulting in the carriers diffusing into the bulk. The dominant recombination mechanism in silicon is Auger, with a characteristic recombination time greater than 6 ps at high carrier densities [LLM01]. This is too long to consider in ultrafast ablation, since in that time the electron-hole pairs will have left the simulation area due to the carrier density gradient. This leaves carrier-phonon scattering, in which the carriers relax by transferring their kinetic energy to the lattice by phonon emission. The characteristic lifetime for emitting optical and acoustic phonons is $\tau_{LO} = 1\,\text{ps}$ and $\tau_{TA} = 10\,\text{ps}$, respectively [HGC98, LLM00b]. In the simulation this is implemented by randomly transferring the electron energy to a atom within a radius of 3.8 Å, by which some delocalization is accounted for.

An ion and electron can recombine if they pass within 1 Å of each other. Considering conduction electrons supplied from the bulk, an ion-lifetime of 500 fs was assumed for the simulations [HGC98].

Chapter 6

Results and Discussion on Thresholds

Simulations have been done for various pulse lengths and fluences. In the following we will present the result and discuss the observations. All simulations were carried out for a single laser pulse at a wavelength of $\lambda = 800\,\text{nm}$. The pulse length was varied from 50 fs to 400 fs (pulse width at 1/e). The initial bulk height was 81.5 nm and 1.06 μm for the MD and HF system, respectively. The lattice cell size in the HF system was 5 nm. The MD system contained 120000 atoms. The simulation volume was 5.4 nm wide (x,y direction) and extended 37 nm above the surface of the bulk. A step size of $\Delta t = 0.5\,\text{fs}$ was used for both MD and HF. The lattice was set to a temperature of $T_0 = 300\,\text{K}$ by initialising the atomic velocities to a Maxwell-Boltzmann distribution, followed by a 200 fs (400 time steps) of Langevin damping to T_0 and an 800 fs relaxation period. The laser pulse is started at $t_0 = 1\,\text{ps}$ with the peak intensity at time $t = t_0 + \tau_L$. Fluences are given as absorbed fluence, unless noted otherwise.

Figure 6.1 shows the lattice temperature at different positions (z) for a pulse length of 100 fs.

The surface is at $z = 81.5\,\text{nm}$, the MD-HF interface is at $z = 0\,\text{nm}$. In 6.1a the fluence is slightly below the ablation threshold ($F_{\text{abs}}^{[a]} = 0.13\,\text{J//cm}^2$), and in (b) it is just above. The lattice peak surface temperature in the latter case is about 8500 Kelvin and is reached 4 ps after the start of the pulse ($t = 5\,\text{ps}$). Cavalleri et al. observed surface temperatures of 2000-3000 K for fluences between the melting and ablation thresholds and 3000-4000 K at the threshold[1] for ablation for both Si[100] and Si[111] surfaces ($\tau_L = 100\,\text{fs}$, $\lambda = 620\,\text{nm}$) [CSTB+99]. This agrees with our result below the threshold (6.1a), but is lower than what we observe for a fluence just above threshold. The surface temperature in fig. 6.1b drops to 3000-4000 K within 10 ps and then falls below the melt temperature at $t \approx 20\,\text{ps}$.

Figure 6.2 shows the pressure evolution in the MD system below the ablation threshold. The shock wave moves at a speed of about 7.8 km/s and is reflected at the MD-HF interface. Cox-Smith et al. measured the sound velocity to be 7.9 km/s in c-Si and 6.3 km/s in a-Si [CSLD85]. Harada et al. have studied the generation of acoustic pulses produced by picosecond laser pulses of fluence just below the threshold for ablation. They measured the velocity of acoustic pulses to be 8.5 km/s, which is close to their observed sound velocity of 8.43 km/s for Si(100) [HKT+89]. Hao et al. report a sound velocity of 8.48 km/s, in agreement with Harada et al. This is in reasonable agreement with our observation. The reflection of the shock wave off the interface is not ideal, and a non-reflecting boundary should be implemented for future investigations. Such a shock-absorbing boundary condition has been developed by Zhigilei et al. [ZG99d, SZG99, SUZG02].

Above the ablation threshold, shortly after the pulse energy is deposited in the system, there is

[1] The threshold was defined as the minimum fluence required for visible crater formation, i.e. by means of plotting the area of the crater vs. fluence and extrapolating to zero.

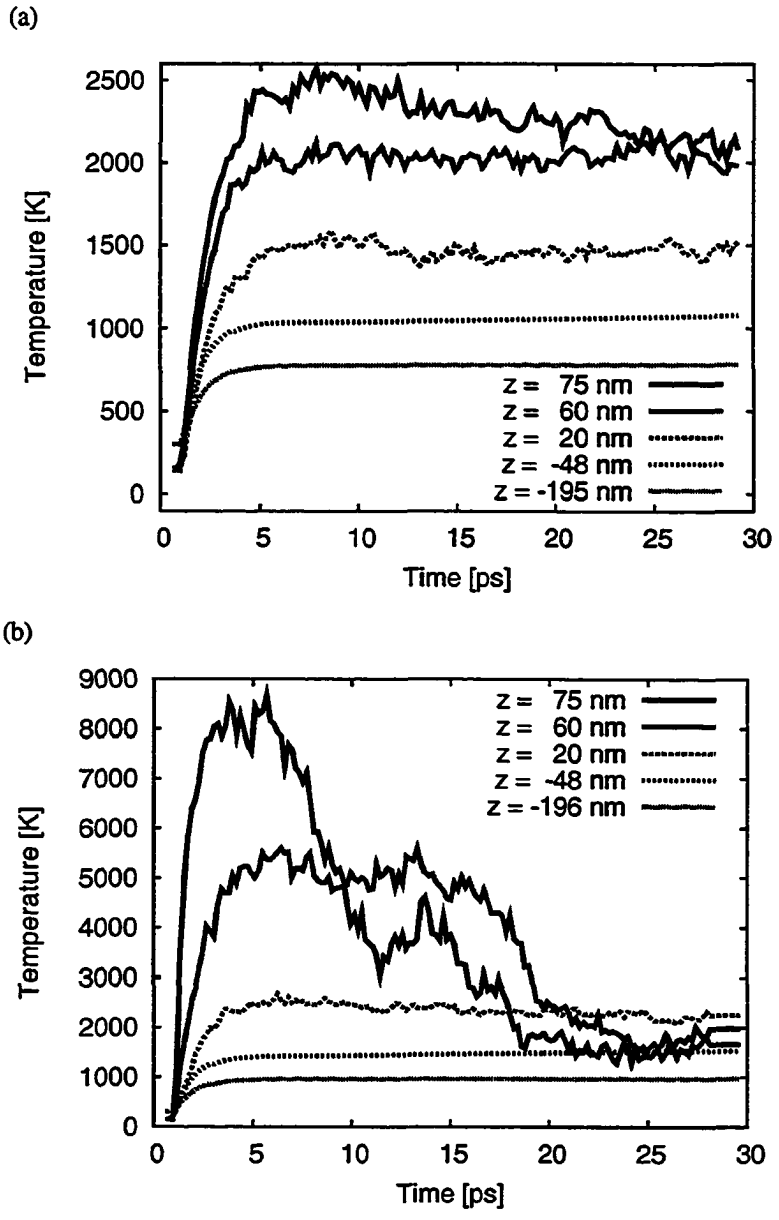


Figure 6.1: Temperature at various depths for pulse length $\tau_L = 100\text{fs}$ ($\lambda = 800\text{nm}$). (a) $F_{\text{abs}} = 0.10\text{J/cm}^2$, (b) $F_{\text{abs}} = 0.16\text{J/cm}^2$.

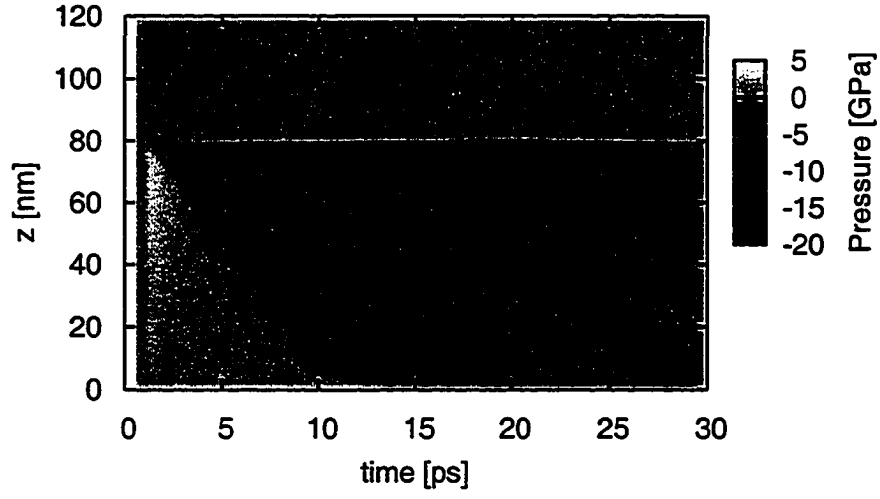
a sharp rise in pressure (figure 6.3). The fast heating of the system does not allow the system to thermally expand quickly enough, resulting in high (compressive) pressures of up to 10 GPa. The relaxation of the pressure results in the expansion of the system. We observe very high tensile (negative) pressures of up to -80 GPa. In comparison, Lorazo et al. report peak pressures of ~10 GPa[2] before expansion of the volume, followed by a relaxation and sign-reversal of the pressure (for $\tau_L = 10\text{ps}$ and $\lambda = 308\text{nm}$) [LLM00a]. This agrees with our results. The compressive pressure reaches a maximum about $t \approx 1.8\text{ps}$ (0.6ps after the peak of the laser pulse), and then becomes tensile (negative) at about $t \approx 2.7\text{ps}$. Lorazo used a much longer pulse of 10 ps and observes a peak compressive pressure at about 8.5 ps after the start of the pulse (3.5 ps after peak of pulse), and a sign-reversal of the pressure 6.5 ps later. Given the longer pulse length used by Lorazo, it is expected that the peak compressive pressure is reached later than in our simulation. The faster relaxation of the pressure in our simulation is due to the "bond breaking" mechanism as described in sec 5.3, which allows the material to melt (non-thermally) and expand quicker.

6.1 Melting

The pair-correlation function $g(r)$ provides a way to analyze the structure of the lattice. It is defined to be proportional to the number of particles separated by distance r, averaged over all directions. The Fourier transform of $g(r)$ gives the experimentally measurable structure factor [Rap95, SW85].

[2]Reported as negative pressure, in opposite sign of our definition (sec. 3.4.1)

(a)



(b)

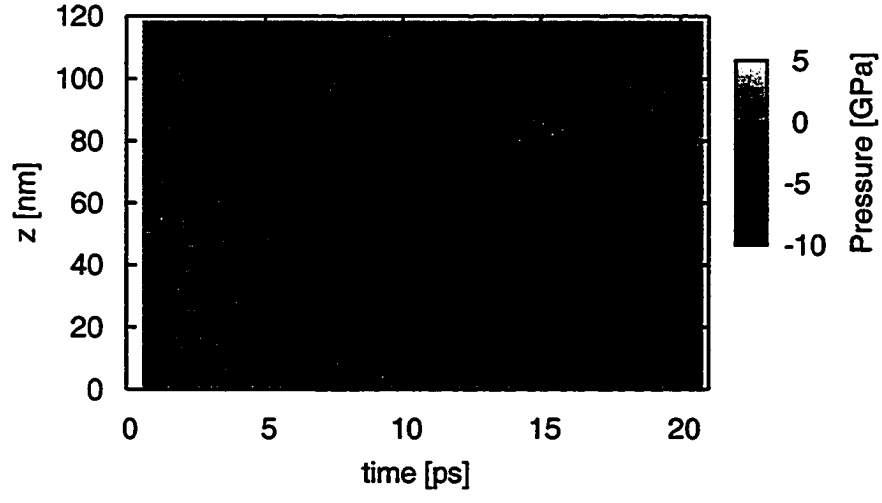


Figure 6.2: Pressure wave due to laser pulse ($\lambda = 800\,\text{nm}$) below the ablation threshold. The wave front moves at a velocity of $\approx 7.8\,\text{km/s}$. (a) $F_{\text{abs}} = 0.06\,\text{J/cm}^2$, $\tau_L = 50\,\text{fs}$ (b) $F_{\text{abs}} = 0.30\,\text{J/cm}^2$, $\tau_L = 800\,\text{fs}$.

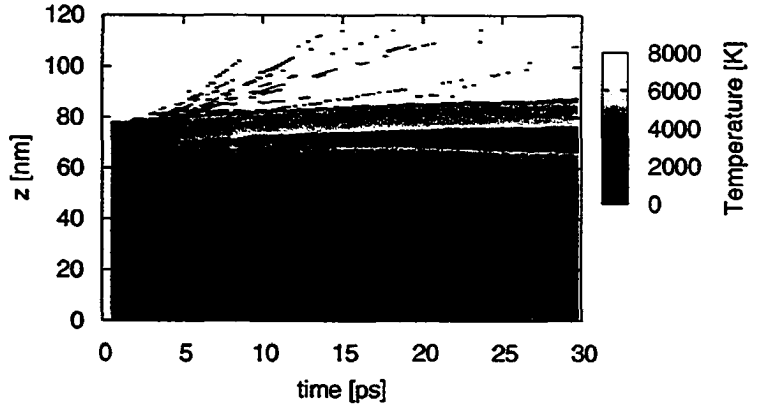
Radial integration over $g(r)$ yields the coordination number (CN) :

$$n(r) = 4\pi\rho \int_0^r s^2 g(s) ds \tag{6.1}$$

where ρ is the density. Evaluating $n(r)$ at the first minimum past the first peak in $g(r)$ (see Fig. 6.4) gives the number of nearest neighbours [SW85].

The coordination number (number of nearest neighbours) for liquid silicon is ~ 6 [WS75, GS79], which is higher than for solid phase (CN=4). Thus by measuring the coordination number we can get an indication of the phase. Figure 6.5a shows the coordination number as a function of position in the bulk (z) and time. The interface between the solid and liquid phase is at $z \approx 55$ nm, which gives a melt depth of 25 nm. The temperature at this depth is 2600-2750 K, which is just above the melting threshold (see fig 3.9). The final melt depth is reached at $\sim$10 ps. The average velocity of the melt front up to that time is estimated at $\sim$2500 m/s. This is a rather crude estimate, since during the first 5-7 ps after the pulse the melt front is not well defined (e.g. see fig. 6.7). The threshold fluence for melting can be estimated from the melt depth (extrapolation to zero depth) or the minimum fluence required to bring the surface temperature above the melting temperature. Table 6.1 gives the estimated melt thresholds. The values were obtained by observing both the melt depth (from coordination numbers) and the surface temperature. Both methods were in agreement within the errors. Borowiec et al., Bonse et al., and Cavalleri et al. present melting thresholds that are approximately half of the ablation threshold (absorbed fluence) [BMWH03, BKWB04, CSTB$^+$99]. Our results do not agree with that ratio, but are closer to 75% of the ablation threshold. The rel-

(a)



(b)

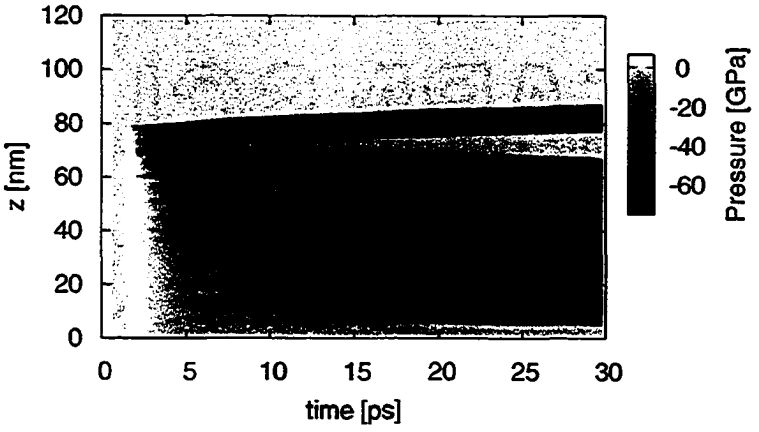


Figure 6.3: Evolution of temperature (a) and pressure (b) for a 400 fs pulse of $0.3\,\mathrm{J/cm^2}$ and wavelength $\lambda = 800\,\mathrm{nm}$. The vertical axis is the position (z) in the material, measured from the MD-HF interface and in the normal direction to the surface. The horizontal axis is the time from the start of the simulation. The laser pulse starts at $t = 1\,\mathrm{ps}$ and reaches peak intensity at $t = 1.4\,\mathrm{ps}$

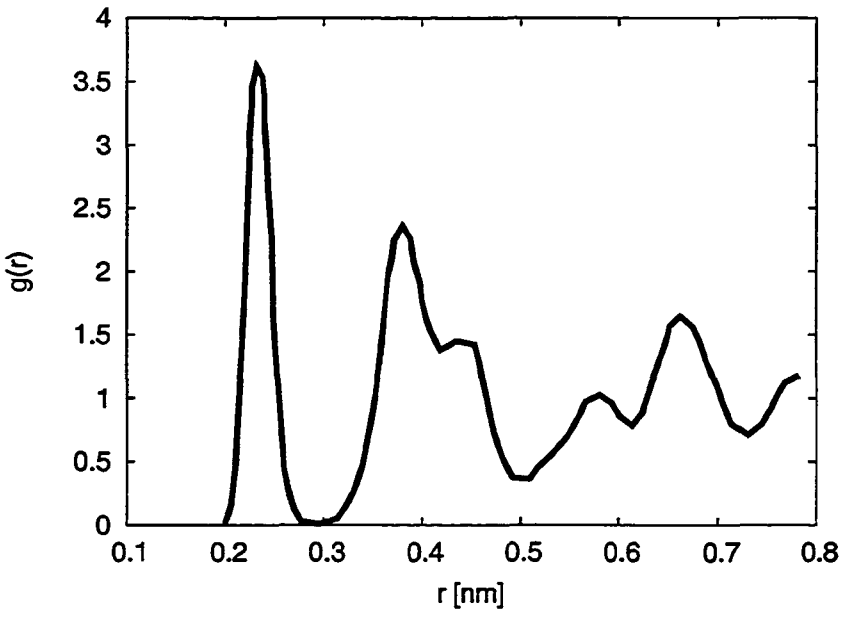


Figure 6.4: Pair-correlation function for Si in the crystalline phase ($T = 2015$K, slightly below melting point) [SW85].

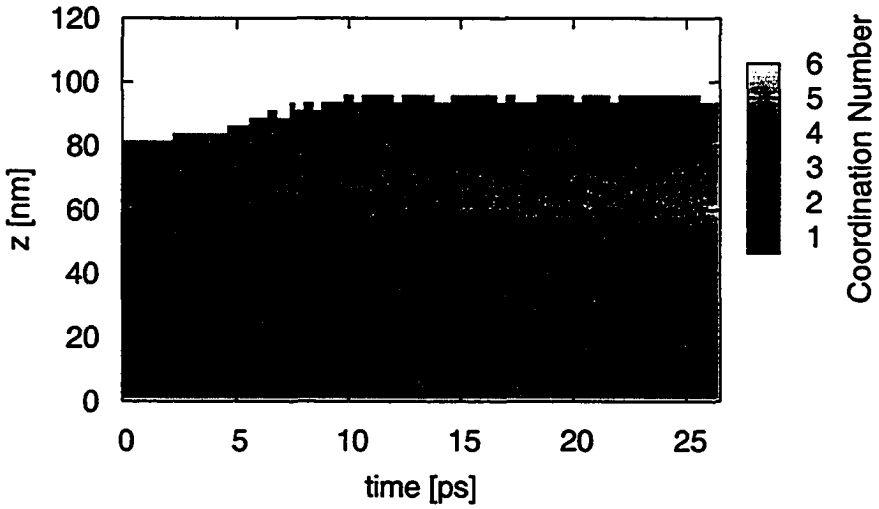
atively low thermal conductivity for silicon in our simulation compared to the true experimental thermal conductivity values may explain some of this discrepancy.

6.2 Ablation

Snapshots in time of the top region of the simulation volume during a typical ablation process are shown in figures 6.6 and 6.7 for a fluence just above the ablation threshold for 400 fs and 100 fs pulses, respectively. The light coloured atoms denote atoms in the ground state, i.e. ones that have not absorbed any photons. The darker particles are excited (absorbed at least one photon) atoms and ions (absorbed energy in excess of 4.85 eV). As outlined in section 5.3, the excited atoms have broken bonds and are therefore less tightly bound. In addition, as the energy is transfered from the electronic system to the lattice by decay of excited states, the lattice heats up. This creates strong pressures in the lattice as outlined above (see figure 6.3). As the system expands and the pressure relaxes, high tensile pressures develop, which lead to nucleation of bubbles (fig. 6.6) and eventually ejection of particles, i.e. ablation. This is consistent with simulation results by Lorazo et al. [LLM03]. The ablation mechanism observed here is termed *phase explosion*: The material heats up and melts isometrically, i.e. on a time scale shorter than the system can expand, giving rise to high pressures. The pressures then relax by mechanical expansion and the system cools adiabatically, as this occurs faster than the time required for significant thermal diffusion. This gives rise to nucleation of gas bubbles in the hot liquid, resulting in a mixture of liquid and gas.

Figure 6.8 shows the number of particles removed. A particle is removed from the simulation if it reaches the top of the simulation volume, which is 37 nm above the initial surface of the bulk.

(a)



(b)

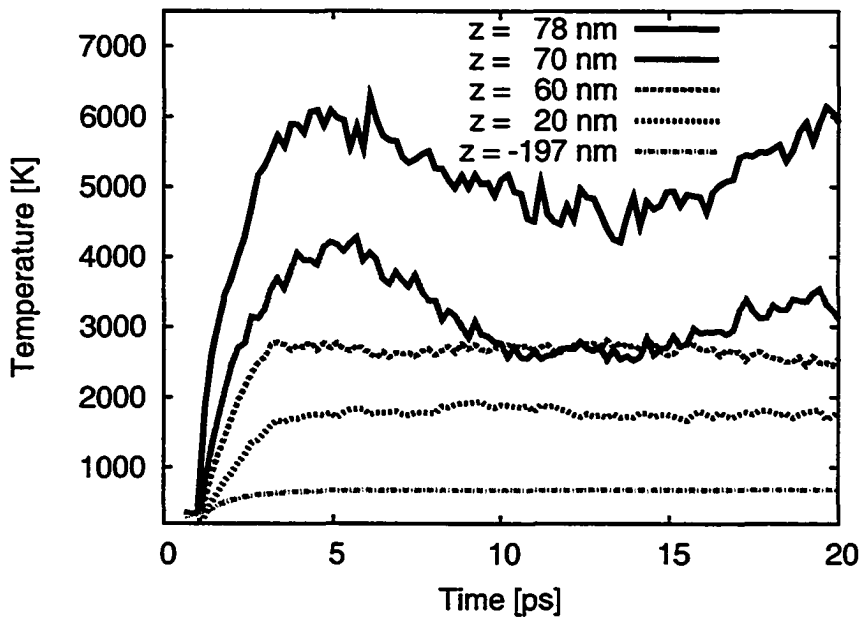


Figure 6.5: Coordination number (a) and temperature (b) for $F_{abs} = 0.10 J/cm^2$ and $\tau_L = 50 fs$.

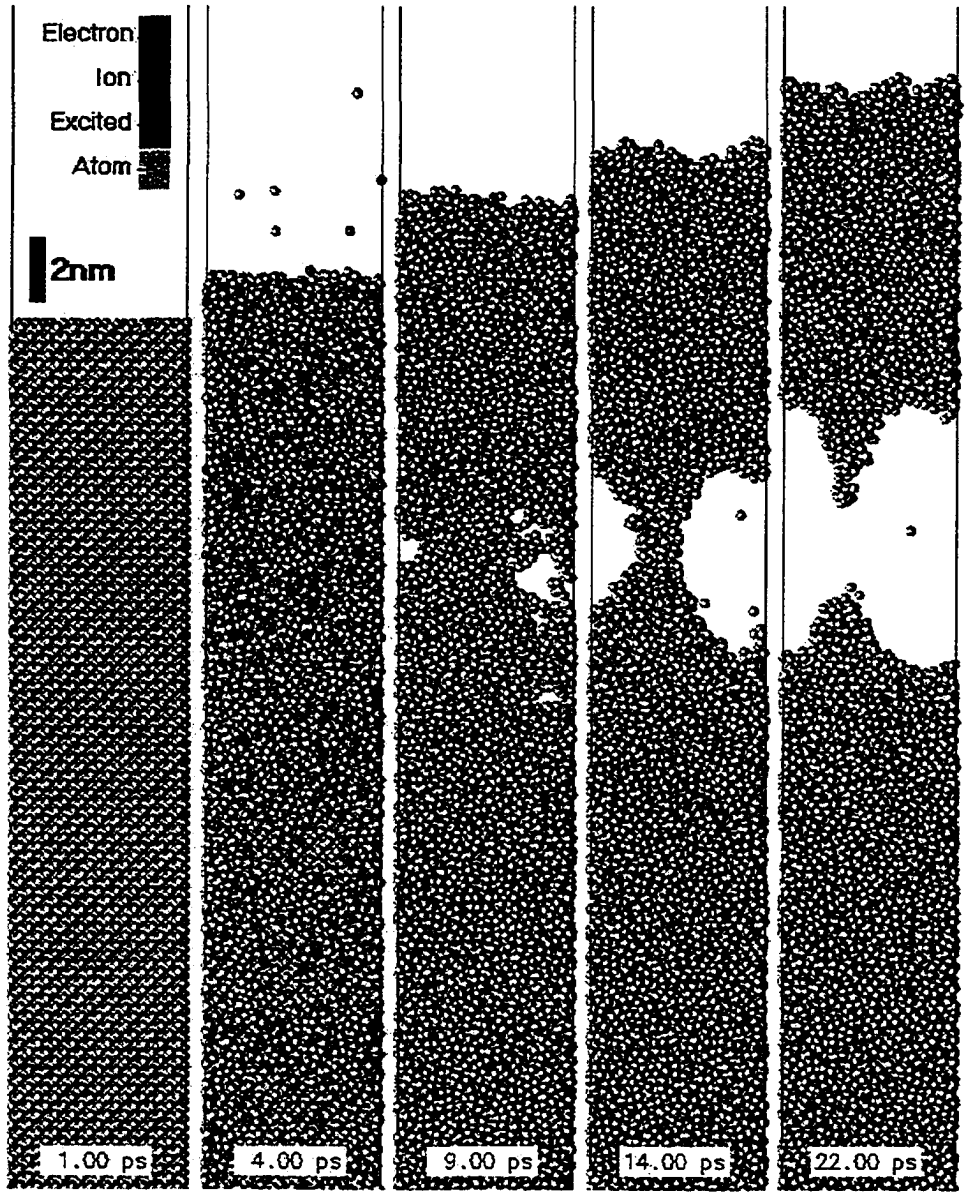


Figure 6.6: Ablation sequence for 400 fs laser pulse (1/e) with a fluence of 0.30 J/cm^2 ($\lambda = 800$ nm). Shown is the top portion of the MD system. The laser pulse starts at $t = 1$ ps and reaches peak intensity at $t = 1.4$ ps

The electrons are ejected from the bulk very quickly. At fluences of $F_{abs} = 0.16 J/cm^2$ or higher, we observe significant particle ejection starting at about 4 ps after the pulse and reaching a peak removal rate at about 12-13 ps after the pulse. There is a small peak visible at $t = 4 ps$, which corresponds to excited atoms and ions.

6.2.1 Thresholds

The threshold for ablation has been obtained for several pulse lengths. The thresholds were obtained by varying the fluence and visually examine the evolution of the system as well as counting the number of atoms removed. The threshold value was taken as the median between highest fluence resulting in no ablation (F_{abs}^{-}) and the lowest fluence giving clearly visible ablation (F_{abs}^{+}), i.e. about 1 atom/Å^2. The fluence increments used for determining the thresholds were $0.01 - 0.02 J/cm^2$, and with these stepsizes ablation would set in suddenly and clearly. If the increments were reduced (more simulation runs would be required), a better definition for the threshold may be required. For now the current one is sufficient. The results are plotted in figure 6.9. The error bars indicate

τ_L (1/e) $[fs]$	τ_L (FWHM) $[fs]$	$F_{abs}^{[m]}$ $[\frac{J}{cm^2}]$	$F_{abs}^{[a]}$ $[\frac{J}{cm^2}]$
50	42	0.07±0.01	0.09±0.01
100	83	0.10±0.02	0.13±0.02
200	167	0.16±0.02	0.19±0.02
400	333	0.24±0.02	0.26±0.03
800	666	0.39±0.03	-

Table 6.1: Threshold absorbed fluences for melting ($F_{abs}^{[m]}$) and ablation ($F_{abs}^{[a]}$). Some of the thresholds have not been established yet and further simulations are required.

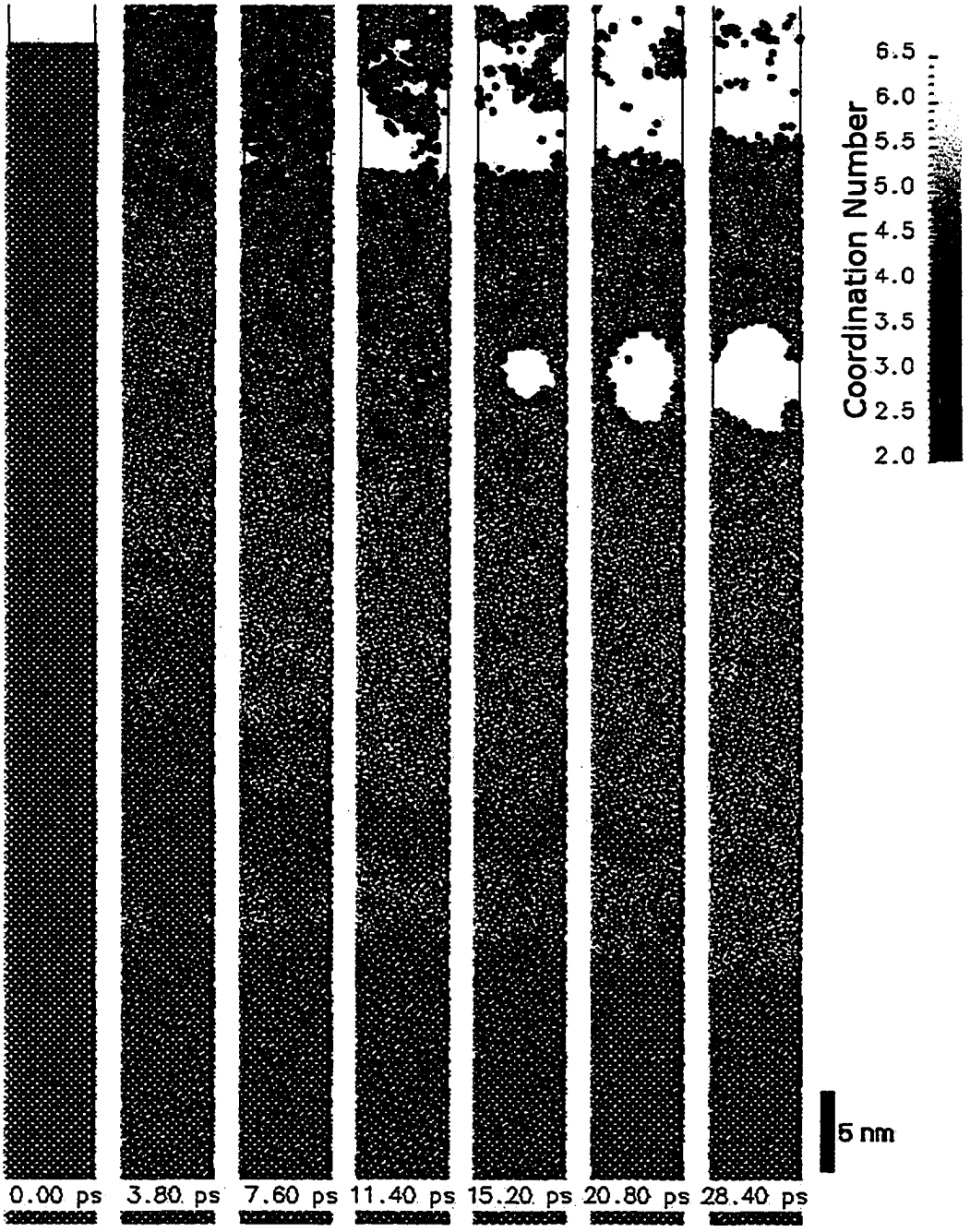


Figure 6.7: Ablation sequence for 100 fs laser pulse (1/e) with a fluence of 0.16 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.1$ ps

the range between the fluences F_{abs}^{-} and F_{abs}^{+}. We observe a strong pulse length dependence. This is mainly due to the varying skin depth. For wavelengths in the near-IR and intensities greater than 10^{12} Watt/cm^2 (e.g. $F_{abs} \approx 0.1$ J/cm^2 and $\tau_L \approx 100$ fs), the light absorption is dominated by non-linear multiphoton absorption. Thus the skin depth depends on the intensity, which in turn is proportional to the pulse length for a given fluence. Assuming that the ablation threshold fluence is proportional to a threshold energy density (at/near the surface), then the threshold fluence should be proportional to the square-root of the pulse length. The argument is as follows:

$$F_{abs}^{[a]} \propto E_{abs}^{[a]} \times d \tag{6.2}$$

where $E_{abs}^{[a]}$ is the minimum absorbed energy per unit volume required for ablation and d is the absorption skin depth. For this estimate we assume that the heat diffusion is negligible (i.e. $d \gg$ diffusion length). Using the non-linear absorption coefficient, we can estimate the skin depth as follows:

$$d \approx (\alpha + \beta I_{peak})^{-1} \approx \frac{\sqrt{\pi}\tau_L}{2\beta F_{abs}}, \qquad \text{(assuming: } \alpha \ll \beta I_{peak}\text{)} \tag{6.3}$$

α is the linear absorption coefficient, β is the two-photon absorption coefficient, I_{peak} is the peak laser intensity, F_{abs} is the absorbed laser fluence, and τ_L is the duration of the Gaussian laser pulse. For comparison, the diffusion length is

$$L = 2\sqrt{\tau_L D} \tag{6.4}$$

where τ_L is the laser pulse length, and D is the thermal diffusivity ($D = 0.8\,\text{cm}^2/\text{s}$ for Si) [Bäu00]. We can express the diffusivity as $D = \kappa/(\rho c_P)$, where κ is the thermal conductivity, ρ is the density

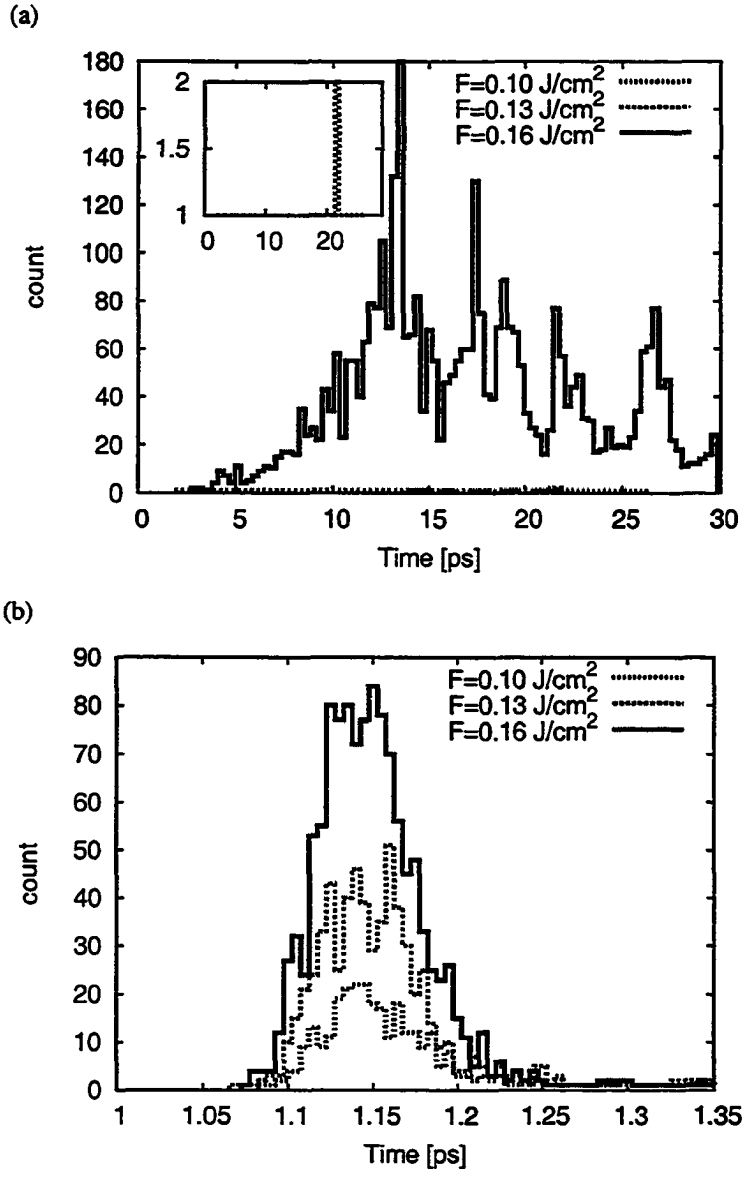


Figure 6.8: Count of ablated atoms (a) and electrons (b), i.e. particles that have reached the top of the simulation volume and have been subsequently removed from the simulation. The pulse length is 100 fs (1/e). The inset in figure (a) shows the count for $F_{abs} = 0.10 J/cm^2$ and $F_{abs} = 0.13 J/cm^2$, where only a few atoms get removed (evaporated).

and c_P is the heat capacity [Bäu00]. Using values determined in section 3.4 for the SW silicon ($\kappa = 0.139 W/(cmK)$, $c_P \approx c_V = 0.951 J/(gK)$), $\rho = 2.329\,\mathrm{g/cm^3}$, we obtain $D = 0.063\,\mathrm{cm^2/s}$. The diffusion lengths and absorption depths (at threshold fluence for ablation, using values given in fig. 6.9) for different pulse lengths are plotted in figure 6.10. It shows that our assumption of $d \gg L$ is reasonable, particularly when considering the diffusivity obtained for the SW potential.

Substituting eqn. 6.3 into eqn. 6.2 and solving for the fluence $F_{\mathrm{abs}}^{[a]}$ gives:

$$F_{\mathrm{abs}}^{[a]} \propto \sqrt{\frac{\sqrt{\pi}}{2\beta} E_{\mathrm{abs}}^{[a]} \tau_L} \tag{6.5}$$

As we can seen in figure 6.9 the threshold values follow the square-root dependence on the pulse length very well. We can fit the absorption threshold to

$$F_{\mathrm{abs}}^{[a]} = 0.01304 \frac{\mathrm{J}}{\mathrm{cm^2}\sqrt{\mathrm{fs}}} \times \sqrt{\tau_L} \tag{6.6}$$

This is consistent with the fact that the heat diffusion length is much shorter than the the absorption length for the ablation regimes investigated. As the pulse length increases to picoseconds and longer, the threshold is expected to deviate from the square-root dependence. Both the absorption length (at threshold fluence) and the diffusion length increase as the square-root of the pulse length (eqn. 6.3 and 6.4), until linear absorption becomes more dominant and heat diffusion starts to play a more important role. Currently, the simulations do not include avalanche ionisation, which may be a significant factor in the absorption of the laser pulse and consequently in the process of laser ablation [PVH+98]. We should note that the error bars in figure 6.9 are quite large and further simulations

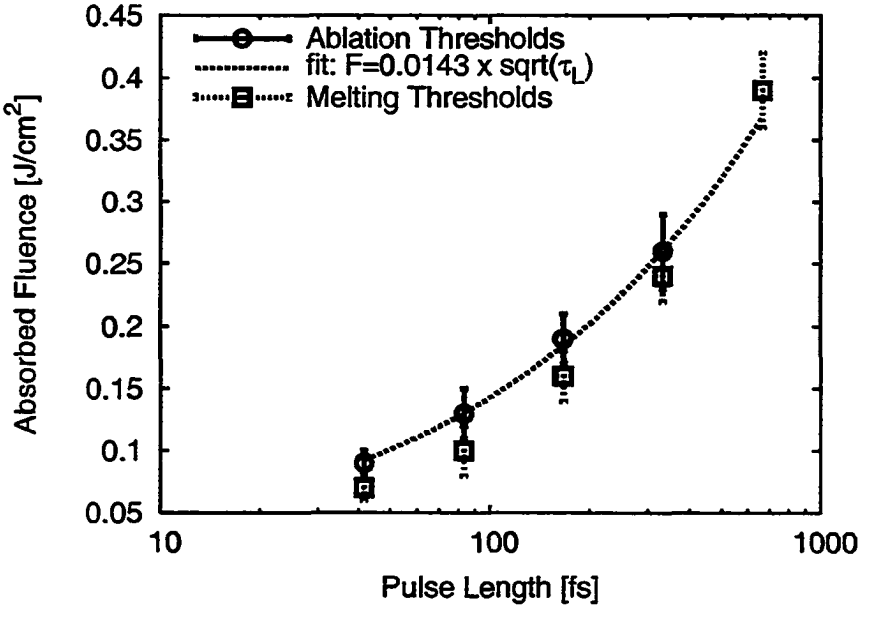


Figure 6.9: Melting and ablation thresholds at different pulse lengths (FWHM).

are necessary to reduce these and to verify the square-root scaling.

At the lower temperatures, the heat capacity is significantly higher for the SW material than for real Si, as noted in section 3.4.2. Thus we would expect our threshold values to slightly overestimate the melting and ablation threshold. With the heat flow model the temperature at the bottom of the MD system is raised, which reduces the temperature gradient and artificial outflow of energy. Without the HF system and the bottom boundary clamped at room temperature the threshold fluence would be slightly higher than what shown in figure 6.9.

6.2.2 Comparison with previously reported results

The ablation thresholds for Si and ultrafast laser pulses are summarised in table 6.2 for theoretical studies, and table 6.3 for experimental values. The column $F^{[a]}$ gives the threshold values as reported in the respective reference. $F_{\text{abs}}^{[a]}$ are the absorbed fluence, taking into account reflectivity. The reflectivity of silicon and linear absorption coefficient are plotted as a function of wavelength in fig. 1.1. Figure 6.12 shows a plot of the threshold values ($F_{\text{abs}}^{[a]}$). We can see that even for the experimental values the various groups report different ablation thresholds. For single-pulse thresholds, the ambient environment plays an important role. If the experiments are done in air then the Si surface will have an oxide layer, which will affect the measured threshold. For multi-shot ablation this would be less of a concern since the oxygen will be removed after the first few shots and the later pulses will see a clean silicon surface. Bonse et al. [BBK+02, BKWB04] report having a 3 nm oxide layer on top of the c-Si [111] surface. Cavalleri et al. [CSTB+99] placed the Si sample in an ultra-high vacuum (10^{-10} torr) for their ablation experiments. Borowiec et al. use a slight vacuum

of ~ 0.1 mbar. They report an uncertainty of 50% in the fluence measurements.

There is a large variation in reported threshold fluences, due to different experimental conditions and different techniques used to define the threshold of ablation. In particular, most groups report threshold for only one pulse length, thus one should be cautious in deducing pulse-length scaling. Only Jeschke et al. [JGL+02] and Pronko et al. [PDS+95, PVS+96, PVH+98] provide thresholds at different pulse lengths. For Jeschke, results are reported for pulse lengths of 20 fs and 500 fs with absolute values that are within 30% of ours. However, their experimental thresholds scale much less with pulse length. Our interpretation of their MD results (as given in table 6.2) show an approximate square-root scaling, however, when estimating the fluences using linear absorption[3] for the skin-depth gives much less scaling (see below for details on fluence estimation). Pronko et al. [PVH+98] give results for pulse lengths from 86 fs to 7 ns (see figures 6.11 and 6.12). As noted before there is significant variation in the measured thresholds reported in literature, and this is also evident in Pronko's results. While our results are within the range of their measured thresholds, we show much stronger pulse length scaling. According to Pronko et al. the dominant absorption mechanism is avalanche ionisation [PVS+96, PVH+98]. Since there the longer pulses can interact via increasing the electron number density through a collisional or avalanche ionisation process, the effective absorption coefficient has much less dependence on the pulse length (or intensity) than in non-linear, two-photon absorption. The next version of our code will include avalanche ionisation and we will investigate its contribution to the ablation process.

Jeschke [JGL+02] provides the ablation threshold in terms of absorbed energy per atom $E_{th,a}$.

[3]Jeschke used the Drude formula (i.e. assuming linear absorption) in another paper to estimate a threshold fluence from his results [JGB01].

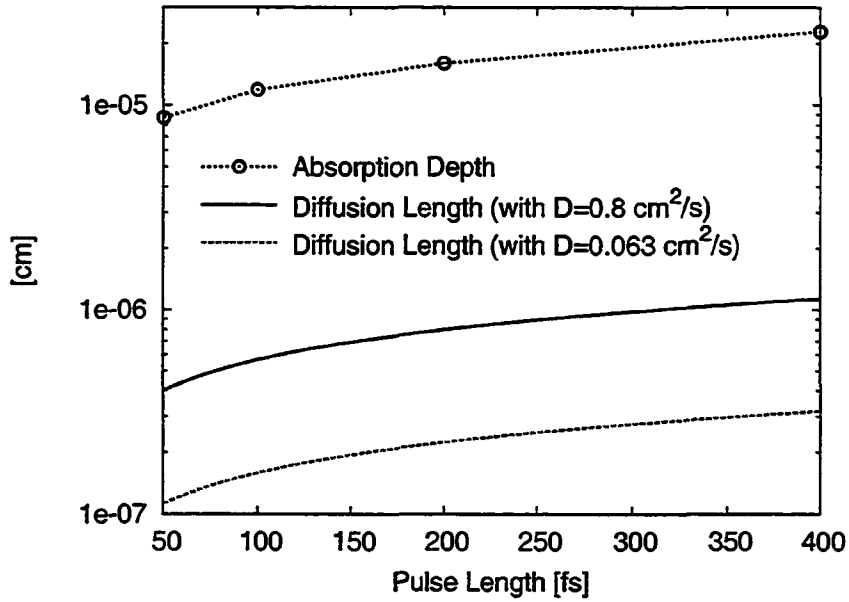


Figure 6.10: Diffusion length (from eqn. 6.4) and absorption depth (from eqn. 6.3) vs. pulse length. The diffusion length is given for both the literature value of the diffusivity ($D = 0.8\text{cm}^2/\text{s}$) and estimated value from thermal properties measured in section 3.4 ($D = 0.063\text{cm}^2/\text{s}$).

Group	τ_L	λ nm	R	α $[\frac{1}{cm}]$	β $[\frac{cm}{GW}]$	$F^{[a]}$ $[\frac{J}{cm^2}]$	$F^{[a]}_{abs}$ $[\frac{J}{cm^2}]$
Ohmura, Watanabe [WIOM00, IWF+98, OM98]	100 fs	266	0.474			<0.01	<0.01
Meunier, Lorazo	500 fs	266	0.733	2.09×10^6	40	0.35	0.093
[LLM01, LLM00b,	10 ps	308	0.591	1.5×10^6	40	0.25-0.30	0.10-0.12
LLM03]	50 ps	266	0.733	2.09×10^6	40	0.45	0.12
Singh, Pronko	2.5 fs	308	0.591	1.5×10^6		0.28	0.115
[PDDS95]	10 fs					0.27	0.110
	100 fs					0.23	0.094
	1.5 ps					0.16	0.066
	100 ps					0.21	0.086
	2 ns					0.78	0.320
[PDDS95]	100 fs	800	0.329	1014		0.24	0.16
	300 fs					0.25	0.17
	1.5 ps					0.32	0.22
	6 ps					0.42	0.28
	20 ps					0.67	0.45
	7 ns					2.29	1.54
Campbell,	10 fs	~700	0.338	3500	55	3±1	3±1
Herrmann	50 fs					4±2	4±2
[HGC97, HGC98]	200 fs					6±2	6±2
	1 ps					6±2	6±2
	5 ps					8±2	8±2
Jeschke, Bonse	20 fs	780	0.33			3.7±0.3 $^{eV}/_{at.}$	0.1
[JGL+02]	500 fs					6.2±0.3 $^{eV}/_{at.}$	0.6

Table 6.2: Single shot ablation thresholds from theoretical studies. τ_L is the laser pulse length (FWHM), λ is the wavelength, R is the reflectivity (as reported by Aspnes and Studna [AS83], see fig. 1.1), α and β are the linear and two-photon absorption coefficients, respectively, and $F^{[a]}$ is the ablation threshold fluence as reported by the authors. $F^{[a]}_{abs}$ is the absorbed fluence (i.e. taking reflection into account).

In another paper [JGB01] Jeschke provides a fluence estimate from this type of result. We now use the same scheme to estimate the ablation threshold in this case:

$$F_{\text{inc}}^{[a]} = \frac{eE_0 n_a d}{1-R-T} \tag{6.7}$$

where E_0 is the absorbed energy, e is the Coulomb constant, $n_a = 5.0\times10^{22}$at./cm^3 is the atomic number density, d is the penetration depth, $R = 0.33$ [AS83] is the reflectivity, and T is the transmission. d is given by $d = \lambda/(4\pi k)$, assuming one-photon absorption, where $k = 0.008$ [AS83] is the extinction coefficient. This gives ablation thresholds of 34J/cm^2 and 57J/cm^2 for pulse length of 20 fs and 500 fs, respectively. If instead we estimate the skin depth as given in eqn. 6.3, we get:

$$F_{\text{inc}}^{[a]} \approx \frac{1}{1-R-T}\sqrt{\frac{n_a\sqrt{\pi}\tau_L E_0}{2\beta}} \tag{6.8}$$

The threshold values obtained this way are given in table 6.3. The experiment by Cavalleri et al. [CSTB$^+$99] was performed using a p-polarised beam illuminating the sample at an incident angle of 45 degrees. This was taken into account for the calculation of the absorbed fluence presented in table 6.3: From the Fresnel equation we get:

$$F_{\text{abs}} = F_{\text{inc}} \times \left(1 - \frac{\tan^2(\theta_1-\theta_2)}{\tan^2(\theta_1+\theta_2)}\right) \times \cos(\theta_1) \tag{6.9}$$

where $\theta_1 = 45°$ is the incident angle and $\theta_2 = \arcsin[(n_1/n_2)\sin\theta_1]$. n_1 and n_2 are the refractive indices of the two media (vacuum and air). The cosine term at the end takes into account the stretching of the focal spot.

Cavalleri et al. show in fig. 3 of [CSTB$^+$99] that atoms are removed and observed experi-

mentally, using a quadrupole mass spectrometer (QMS), well below the ablation threshold down to about 60% of the ablation threshold. The particle removal below the threshold is attributed to desorption (evaporation and sublimation). Thus the ablation threshold cannot be arbitrarily defined by the start of removal of atoms, but must be quantified by the number of atoms removed. We have in our study used a value of 1 atom/Å^2, which corresponds to removal of a surface layer about 2 nm thick.

The total number of removed atoms at the end of a 30 ps simulation is given in table 6.4 for different fluences and pulse lengths. In comparing the thresholds with experimental results, the detection limit of the technique and apparatus is of importance. The observable resolution in the number of removed particles affects the measured threshold value. By plotting the number of removed particles in the simulation as a function of fluence it can potentially allow for more realistic comparison with experimental results. Further simulations would be required for this.

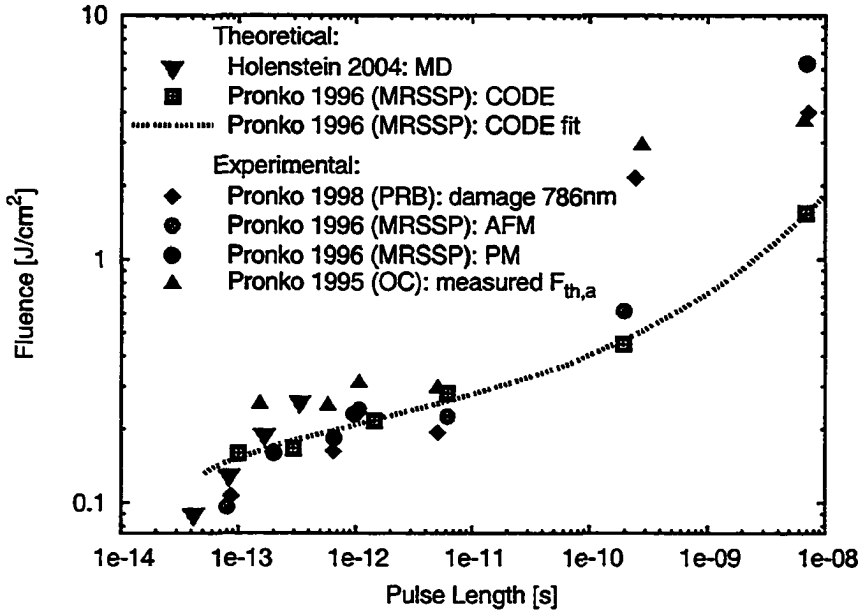


Figure 6.11: Single shot ablation thresholds (absorbed fluence) at different pulse lengths (FWHM) by Pronko et al. [PDS+95, PVS+96, PVH+98]. Measurements were done by examining the area of damage and extrapolating to zero, as well as using atomic force microscopy (AFM) to examine damage due to vaporisation and a photomultiplier to detect onset of plasma emission. Also shown are results from a 2-temperature heat flow model (CODE) that was fit to the data points from the AFM and PM measurements.

Group	τ_L	λ nm	R	$F^{[a]}$ $[\frac{J}{cm^2}]$	$F_{abs}^{[a]}$ $[\frac{J}{cm^2}]$
Bonse [BBK+02]	5 fs	780	0.330	0.20±0.05	0.13
[JGL+02]	(25±5) fs	780	0.330	0.17±0.015	0.114
[BKWB04]	130 fs	800	0.329	0.520	0.349
[JGL+02]	(400±30) fs	780	0.330	0.28±0.03	0.188
Borowiec [BMWH03]	130 fs	800	0.329	0.30	0.20
Cavalleri, von der Linde [CSTB+99]	110 fs	620	0.351	0.300	0.164
Coyne [CMM+04]	150 fs	775	0.331	0.45	0.30
Pronko [PDDS95]	150 fs	800	0.329	0.38	0.25
	580 fs			0.37	0.25
	1 ps			0.46	0.31
	5 ps			0.44	0.29
	280 ps			4.35	2.92
	6.6 ns			5.46	3.66
[PVS+96] (AFM)	80 fs	800	0.329	0.14	0.096
	1 ps			0.36	0.24
	6 ps			0.34	0.23
	200 ps			0.91	0.61
[PVS+96] (PM)	200 fs	800	0.329	0.24	0.16
	650 fs			0.27	0.18
	980 fs			0.34	0.23
	7 ns			9.46	6.35
[PVH+98]	85 fs			0.16	0.11
	200 fs			0.24	0.16
	650 fs			0.24	0.16
	1 ps			0.36	0.23
	5 ps			0.29	0.19
	250 ps			3.21	2.15
	7 ns			5.96	4.00

Table 6.3: Single shot ablation thresholds from experimental studies. τ_L is the laser pulse length (FWHM), λ is the wavelength, R is the reflectivity (as reported by Aspnes and Studna [AS83], see fig. 1.1), and $F^{[a]}$ is the ablation threshold fluence as reported by the authors. $F_{abs}^{[a]}$ is the absorbed fluence (i.e. taking reflection into account).

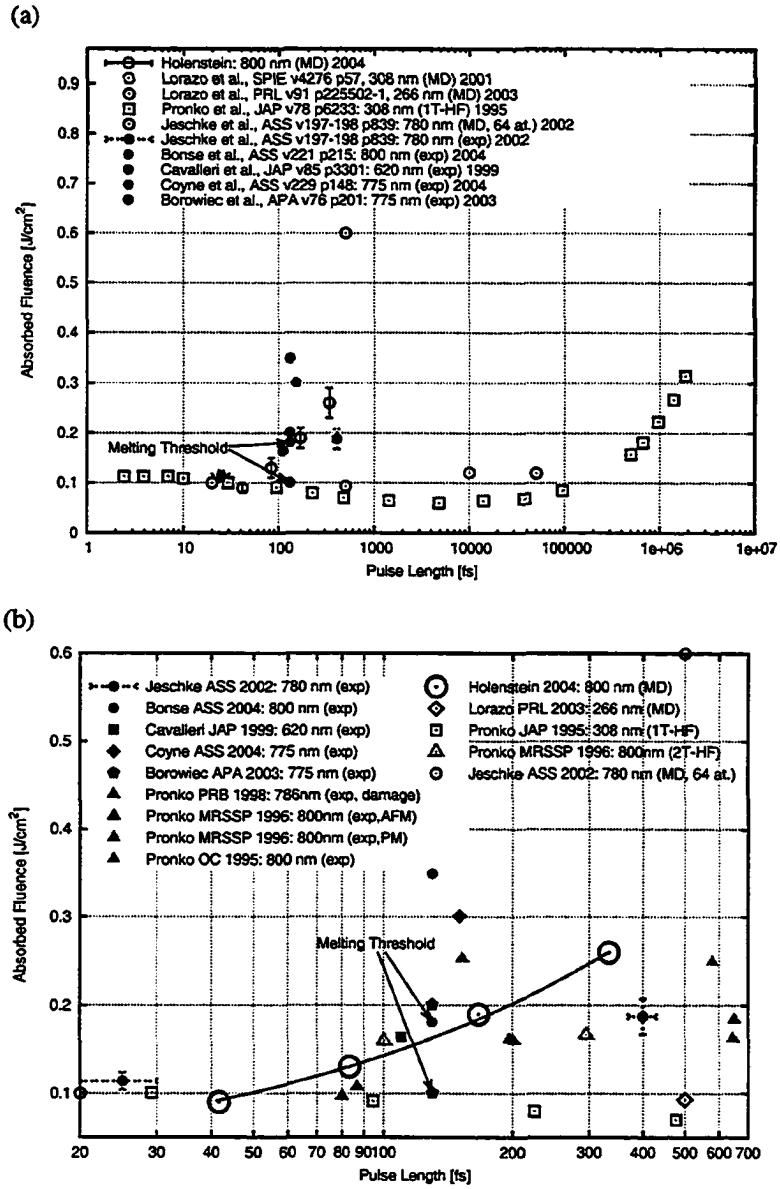


Figure 6.12: Single shot ablation thresholds at different pulse lengths (FWHM) compared to literature values. (a) over the pulsewidth range of 1 fs to 10 ns and (b) over the range of 20 fs to 700 fs. The solid (filled) symbols represent experimental values and the open ones are theoretical values.

τ_L (1/e) $[fs]$	F_{abs} $\left[\frac{J}{cm^2}\right]$	# of atoms removed
50	0.06	0
50	0.08	4
50	0.10	52
100	0.10	2
100	0.13	22
100	0.16	3608
200	0.18	7
200	0.20	9
200	0.24	10030
400	0.22	0
400	0.26	0
400	0.30	22

Table 6.4: Number of removed atoms by the end of a 30 ps simulation versus fluence for different pulse lengths.

Chapter 7

Conclusions

A simulation code has been developed to model ultrafast laser ablation of silicon. The molecular dynamics technique was employed for the top portion of the simulation volume and a heat flow model used to couple the MD system to an infinite bulk medium. The ablation thresholds were determined for pulse lengths between 50 fs and 400 fs.

By modelling only the central part of the laser spot and using periodic boundary conditions in the transverse direction to the incident laser pulse, we were able to simulate a surface layer of up to ~0.1 micron with MD and several microns with HF. The change of boundary condition from a heat bath in the sides, which was used in the initial code, to periodic boundaries improved the threshold results by an order of magnitude. The use of a heat flow model to extend the simulation volume, at little additional computational cost, provided a more realistic model. This improved the temperature gradient and removed artificial cooling effects that would result from coupling to the MD system directly to the heat bath at room temperature. Such excessive cooling result in higher thresholds for

melting and ablation.

In order to speed up the simulations the code has been parallelized using MPI. An attempt to further improve the simulation speed a lookup table for the Stillinger-Weber force and potential was implemented. This scheme turned out to be several times slower than dynamically calculating the force and potential.

An algorithm has been developed to thermally couple the MD system with the HF system. The particles at the bottom of the molecular dynamics system are damped to the temperature at the top of the HF system using Langevin dynamics. Similarly, the heat transfer from the MD system to the HF system is computed from the Langevin equations.

As the laser light is absorbed in the bulk, the temperature rises. This happens very fast, within a few picoseconds, and the system does not have time to expand quickly enough, resulting in large compressive pressure on the order of 10 GPa. Upon relaxation of the pressure by the system expanding, the pressure becomes strongly negative (tensile). These large tensile pressures are the driving force in the ablation process.

It was found that the ablation threshold varies significantly with pulse length. A square-root scaling is observed. In the near-IR and for pulse intensities in the regime we investigated, the two-photon absorption skin depth is large and non-linear absorption is dominant. The results suggest that the heat diffusion length is significantly less than the absorption length.

The calculated ablation threshold, $F_{abs}^{[a]} = 0.13$ J/cm^2, at 100 fs is in approximate agreement with experimental values for pulse lengths on the order of 100 fs, but the square root pulse length scaling and values at longer pulse lengths start to deviate from the experimental results.

However, the effects of avalanche ionization have not been included in the present code and could have significant effect for longer pulses where there is sufficient time for the electron density to grow and effect the absorption. Such an avalanche ionization mechanism should be added to the code.

A significant spread in the experimental results is observed. This is due to different experimental conditions and techniques used. Particularly the detection limit of the technique and apparatus is of importance, as the number of removed particles that can be observed affects the definition for ablation. Plotting the number of removed particles as a function of fluence potentially allows for realistic comparison with experiment, accounting for varying detection limits in experimental techniques.

Further investigations are required to test the observed threshold scaling over a wider range of pulse length. A shock absorbing boundary condition should be implemented in order to avoid reflection of the shock wave at the MD-HF interface. It has not yet been determined to what extend this reflection affects the threshold values, though it is reasonable to assume that with a shock absorbing boundary the threshold would be slightly higher.

Bibliography

[AB87] P.B. Allen and J.Q. Broughton, *Electrical conductivity and electronic properties of liquid silicon*, J. Phys. Chem. **91** (1987), 4964.

[ACD+01] Yu.V. Afanasiev, B.N. Chichkov, N.N. Demchenko, V. A. Isakov, and I.N. Zavestovskaya, *Ablation of metals by ultrashort laser pulses: theoretical modeling and computer simulation*, EPS Conf. on Contr. Fusion and Plasma Phys. ECA, vol. 25A, 2001, p. 2021.

[AD76] S.A. Adelman and J.K. Doll, *Generalized langevin equation approach for atom/solid-surface scattering: General formulation for classical scattering off harmonic solids*, J. Chem Phys. **64** (1976), 2375.

[Aga84] Agassi, *Phenomenological model for picosecond-pulse laser annealing of smiconductors*, J. Appl. Phys. **55** (1984), 4376.

[AKPF94] A. Alavi, J. Kohanofff, M. Parrinello, and D. Frenkel, *Ab initio molecular dynamics with excited electrons*, Phys. Rev. Lett. **73** (1994), 2599.

[AS83] D.E. Aspnes and A.A. Studna, *Dielectric functions and optical parameters of Si, Ge, GaP, GaAs, GaSb, InP, InAs, and InSb from 1.5 to 6 ev*, Phys. Rev. B **27** (1983), 985.

[AS92] Vasilios Alexiades and Alan D. Solomon, *Mathematical modeling of melting and freezing processes*, Hemisphere Publishing Corporation, Washington, 1992.

[AW57] B.J. Alder and T.E. Wainwright, *Phase transition for a hard sphere system*, J. Chem. Phys. **27** (1957), 1208.

[Bai99] R. Baierlein, *Thermal physics*, Cambridge University Press, Cambridge, 1999.

[Bas87] M.I. Baskes, *Application of the embedded-atom method to covalent materials: A semiempirical potential for silicon*, Phys. Rev. Lett. **59** (1987), 2666.

[Bas92] ———, *Modified embedded-atom potentials for cubic materials and impurities*, Phys. Rev. B **46** (1992), 2727.

[Bäu00] D. Bäuerle, *Laser processing and chemistry*, 3rd ed., Springer-Verlag, Berlin, 2000.

[BBK+02] J. Bonse, S. Budach, J. Krüger, W. Kautek, and M. Lenzer, *Femtosecond laser ablation of silicon – modification thresholds and morphology*, Appl. Phys. A **74** (2002), 19.

[BD97] William E. Boyce and Richard C. DiPrima, *Elementary differential equations and boundary value problems*, 6th ed., John Wiley & Sons, Inc., New York, 1997.

[BH85] R. Biswas and D.R. Hamann, *Interatomic potentials for silicon structural energies*, Phys. Rev. Lett. **55** (1985), 2001.

[BHT92] H. Balamane, T. Halicioglu, and W.A. Tiller, *Comparative study of silicon empirical interatomic potentials*, Phys. Rev. B **46** (1992), 2250.

[BKWB04] J. Bonse and A.J. Meixner K.-W. Brzezinka, *Modifying single-crystalline silicon by femtosecond laser pulses: an analysis by micro raman spectroscopy, scanning laser microscopy and atomic force microscopy*, Appl. Surf. Sci. **221** (2004), 215.

[BMWH03] A. Borowiec, M. MacKenzie, G.C. Weatherly, and H.K. Haugen, *Transmission and scanning electron microscopy studies of single femtosecond-laser-pulse ablation of silicon*, Appl. Phys. A **76** (2003), 201.

[BNW89] M.I. Baskes, J.S. Nelson, and A.F. Wright, *Semiempirical modified embedded-atom potentials for silicon and germanium*, Phys. Rev. B **40** (1989), 6085.

[BSR+04] N.M. Bulgakova, R. Stoian, A. Rosenfeld, I.V. Hertel, and E.E.B. Campbell, *Electronic transport and consequences for material removal in ultrafast pulsed laser ablation of materials*, Phys. Rev. B **69** (2004), 054102.

[CFL67] R. Courant, K. Friedrich, and H. Lewy, *On partial difference equations of mathematical physics*, IBM J. of Res. and Devel. **11** (1967), 215.

[CMM+04] E. Coyne, J.P. Magee, P. Mannion, G.M. O'Connor, and T.J. Glynn, *Characterization of laser ablation of silicon using a gaussian wavefront and computer generated wavefront reconstruction*, Appl. Surf. Sci. **229** (2004), 148–160.

[Cow88] E.R. Cowley, *Lattice dynamics of silicon with empirical many-body potentials*, Phys. Rev. Lett. **60** (1988), 2379–2381.

[CP85] R. Car and M. Parrinello, *Unified approach for molecular dynamics and density-functional theory*, Phys. Rev. Lett. **55** (1985), 2471.

[CSLD85] I.R. Cox-Smith, H.C. Liang, and R.O. Dillon, *Sound velocity in amorphous films of germanium and silicon*, J. Vac. Sci. Tech. A **3** (1985), 674.

[CSTB+99] A. Cavalleri, K. Sokolowski-Tinten, J. Bialkowski, M. Schreiner, and D. von der Linde, *Femtosecond melting and ablation of semiconductors studied with time of flight mass spectroscopy*, J. Appl. Phys. **85** (1999), 3301.

[DC98] D.W. Dean and J.R. Chelikowsky, *First principles calculation of the thermodynamic properties of silicon clusters*, Theor. Chem. Acc. **99** (1998), 18.

[FP96] K. Fushinobu and L.M. Phinney, *Ultrashort-pulse laser heating of silicon to reduce microstructure adhesion*, Int. J. Heat Mass Transfer **39** (1996), 3181.

[Gri99] D.J. Griffiths, *Introduction to electrodynamics*, 3rd ed., Prentice-Hall, Inc., New Jersey, 1999.

[GS79] J.P. Gabathuler and S. Steeb, *Über die Struktur von Si-, Ge-, Sn- und Pb-Schmelzen*, Z. Naturforsch. A **34** (1979), 1314.

[Hai97] J.M. Haile, *Molecular dynamics simulation: Elementary methods*, John Wiley & Sons, Inc., New York, 1997.

[HGC97] R.F.W. Herrmann, J. Gerlach, and E.E.B. Campbell, *Molecular dynamics simulation of laser ablation of silicon*, Nucl. Instr. & Meth. in Phys. Res. B, vol. 122, 1997, p. 401.

[HGC98] ———, *Ultrashort pulse laser ablation of silicon: an MD simulation study*, Appl. Phys. A **66** (1998), 35.

[HIM95] H. Haberland, Z. Insepov, and M. Moseler, *Molecular-dynamics simulation of thin-film growth by energetic cluster impact*, Phys. Rev. B **51** (1995), 11061.

[HKT+89] Y. Harada, Y. Kanemitsu, Y. Tanaka, N. Nakano, H. Kuroda, and K. Yamanaka, *Picosecond laser generation of ultrashort acoustic pulses in silicon*, J. Phys. D **22** (1989), 569.

[HMM04] D. Hamelberg, J. Mongan, and J.A. McCammon, *Accelerated molecular dynamics: A promising and efficient simulation method for biomolecules*, J. Chem. Phys. **120** (2004), 11919.

[IMM+98] M. Ishimaru, S. Munetoh, T. Motooka, K. Moriguchi, and A. Shintani, *Molecular-dynamics studies on defect-formation processes during crystal growth of silicon from melt*, Phys. Rev. B **58** (1998), 12583.

[Iof] *New semiconductor materials: Characteristics and properties*, Online, Retrieved July 5, 2004, from http://www.ioffe.rssi.ru/SVA/NSM/Semicond/Si/bandstr.html.

[IWF+98] Y. Ishizaka, K. Watanabe, I. Fukumoto, E. Ohmura, and I. Miyamoto, *Three-dimensional molecular dynamics simulation on laser materials processing of silicon*, Proc. of Laser Materials Processing - ICALEO, vol. 1, 1998, p. A55.

[JGB01] H.O. Jeschke, M.E. Garcia, and K.H. Bennemann, *Theory for the ultrafast ablation of graphite films*, Phys. Rev. Lett. **87** (2001), 015003.

[JGL+02] H.O. Jeschke, M.E. Garcia, M. Lenzner, J. Bonse, J. Krüger, and W. Kautek, *Laser ablation thresholds of silicon for different pulse durations: theory and experiment*, Appl. Surf. Sci. **197-198** (2002), 839–844.

[Kit96] Charles Kittel, *Introduction to solid state physics*, 7th ed., John Wiley & Sons, Inc., Toronto, 1996.

[Kub57] R. Kubo, *Statistical-mechanical theory of irreversible processes: 1. general theory and simple applications to magnetic and conduction problems*, J. Phys. Soc. Jpn. **12** (1957), 570.

[Kub66] ———, *The fluctuation-dissipation theorem*, Rep. Prog. Phys. **29** (1966), 255.

[KUO00] K. Kakimoto, T. Umehara, and H. Ozoe, *Molecular dynamics analysis on diffusion of point defects*, J. of Crystal Growth **210** (2000), 54.

[Lee98] Y.H. Lee, *Silicon di-interstitial in ion-implanted silicon*, Appl. Phys. Lett. **73** (1998), 1119.

[LLM00a] P. Lorazo, L.J. Lewis, and M. Meunier, *Molecular-dynamics simulations of picosecond pulsed laser ablation and desorption of silicon*, Proc. SPIE, vol. 3935, 2000, p. 66.

[LLM00b] ———, *Picosecond pulsed laser ablation of silicon: a molecular-dynamics study*, Appl. Surf. Sci. **168** (2000), 276.

[LLM01] ———, *Simulation of picosecond pulsed laser ablation of silicon: the molecular-dynamics thermal-annealing model*, Proceedings of SPIE **4276** (2001), 57.

[LLM03] ———, *Short-pulse laser ablation of solids: From phase explosion to fragmentation*, Phys. Rev. Lett. **91** (2003), 225502.

[LLR+88] W.D. Luedtke, U. Landman, M.W. Ribarsky, R.N. Barnett, and C.L. Cleveland, *Molecular-dynamics simulations of epitaxial crystal growth from the melt. II. Si(111)*, Phys. Rev. B **37** (1988), 4647.

[LLW+97] C.L. Liu, J.N. Leboeuf, R.F. Wood, D.B. Geobegan, J.M. Donato, K.R. Chen, and A.A. Puretzky, *Computational modeling of physical processes during laser ablation*, Mater. Sci. Eng. B **47** (1997), 70.

[MB91] D. Maroudas and R.A. Brown, *Analysis of point-defect diffusion and drift in cubic-type lattices: Constitutive modeling*, Phys. Rev. B 44 **44** (1991), 2567.

[MB92] ———, *Atomistic calculation of the self-interstitial diffusivity in silicon*, Phys. Rev. Lett. **62** (1992), 172.

[MB93] ———, *Calculation of thermodynamic and transport properties of intrinsic point defects in silicon*, Phys. Rev. B **47** (1993), 15562.

[NBG+03] G.S. Nolas, M. Beekman, J. Gryko, Jr. G.A. Lamberton, T.M. Tritt, and P.F. McMillan, *Thermal conductivity of elemental crystalline silicon clathrate Si_{136}*, Appl. Phys. Lett. **82** (2003), 910.

[NC92] J. Narayan and X. Chen, *Laser patterning of diamond films*, J. of Appl. Phys. **71** (1992), 3795–3801.

[NHR96] José C. Noya, Carlos P. Herrero, and Rafael Ramíres, *Thermodynamic properties of c-Si derived by quantum path-integral monte carlo simulations*, Phys. Rev. B **53** (1996), 9869.

[Nie95] M.H. Niemz, *Threshold dependence of laser-induced optical breakdown on pulse duration*, Appl. Phys. Lett. **66** (1995), 1181.

[OM98] E. Ohmura and I. Miyamoto, *Molecular dynamics simulation of laser ablation phenomena*, Rev. of Laser Eng. **26** (1998), 800.

[PB94] R.H. Poetzsch and H. Boettger, *Interplay of disorder and anharmonicity in heat conduction: Molecular-dynamics study*, Phys. Rev. B **50** (1994), 15757.

[PDDS95] P.P. Pronko, S.K. Dutta, D. Du, and R.K. Singh, *Thermophysical effects in laser processing of materials with picosecond and femtosecond pulses*, J. Appl. Phys **78** (1995), 6233.

[PDS+95] P.P. Pronko, S.K. Dutta, J. Squier, J.V. Rudd, D. Du, and G. Mourou, *Machining of sub-micron holes using a femtosecond laser at 800 nm*, Opt. Comm. **114** (1995), 106.

[PL02] D. Perez and L.J. Lewis, *Ablation of solids under femtosecond laser pulses*, Phys. Rev. Lett. **89** (2002), 255504.

[PMK94] A. Peterlongo, A. Miotello, and R. Kelly, *Laser-pulse sputtering of aluminum: Vaporization, boiling, superheating, and gas-dynamic effects*, Phys. Rev. E **50** (1994), 4716.

[PVH+98] P.P. Pronko, P.A. VanRompay, C. Horvth, F. Loesel, T. Juhasz, X. Liu, and G. Mourou, *Avalanche ionization and dielectric breakdown in silicon with ultrafast laser pulses*, Phys. Rev. B **58** (1998), 2387.

[PVS+96] P.P. Pronko, P.A. VanRompay, R.K. Singh, F. Qian, D. Du, and X. Liu, *Laser induced avalanche ionization and electron-lattice heating of silicon with intense near IR femtosecond pulses*, Mat. Res. Soc. Symp. Proc., vol. 397, 1996, p. 45.

[Rap95] D.C. Rapaport, *The art of molecular dynamics simulation*, Cambridge University Press, Cambridge, 1995.

[RKL+02] C.L. Rountree, R.K. Kalia, E. Lidorikis, A. Nakano, L. Van Brutzel, and P. Vashishta, *Atomistic aspects of crack propagation in brittle materials: Multimillion atom molecular dynamics simulations*, Annu. Rev. Mater. Res. **32** (2002), 377.

[SAPF96] P.L. Silvistrelli, A. Alavi, M. Parrinello, and D. Frenkel, *Ab initio molecular dynamics simulation of laser melting of silicon*, Phys. Rev. Lett. **77** (1996), 3149.

[SAPF97] ———, *Structural, dynamical, electronic, and bonding properties of laser-heated silicon: An ab initio molecular dynamics study*, Phys. Rev. B **56** (1997), 3806.

[SB94] P. Stampfli and K.H. Bennemann, *Time dependence of the laser-induced femtosecond lattice instability of Si and GaAs: Role of longitudinal optical distortions*, Phys. Rev. B **49** (1994), 7299.

[SBN89] J.G. Swadener, M.I. Baskes, and M. Nastasi, *Molecular dynamics simulation of brittle fracture in silicon*, Phys. Rev. B **40** (1989), 6085.

[STBv95] K. Sokolowski-Tinten, J. Bialkowski, and D. von der Linde, *Ultrafast laser-induced order-disorder transitions in semiconductors*, Phys. Rev. B **51** (1995), 14186.

[SUZG02] C. Schäfer, H.M. Urbassek, L.V. Zhigilei, and B.J. Garrison, *Pressure-transmitting boundary conditions for molecular-dynamics simulations*, Comp. Mat. Sci. **24** (2002), 421.

[SW85] F.H. Stillinger and T.A. Weber, *Computer simulation of local order in condensed phases of silicon*, Phys. Rev. B **31** (1985), 5262.

[SYH83a] C.V. Shank, R. Yen, and C. Hirlimann, *Femtosecond-time-resolved surface structural dynamics of optically excited silicon*, Phys. Rev. Lett. **51** (1983), 900.

[SYH83b] ———, *Time-resolved reflectivity measurements of femtosecond-optical-pulse-induced phase transitions in silicon*, Phys. Rev. Lett. **50** (1983), 454.

[SZG99] Julia A. Smirnova, L.V. Zhigilei, and B.J. Garrison, *A combined molecular dynamics and finite element method technique applied to laser induced pressure wave propagation*, Comp. Phys. Comm. **118** (1999), 11.

[Ter88] J. Tersoff, *New empirical approach for the structure and energy of covalent systems*, Phys. Rev. B **37** (1988), 6991.

[TLFC00] Y.Y. Tsui, C. Li, R. Fedosejevs, and C.E. Capjack, *Interaction of femtosecond laser pulses with metals*, Proc. SPIE, vol. 4087, 2000, p. 1201.

[TMTM88] S. Tsuneyuki, H. Akoi M. Tsukada, and Y. Matsui, *First-principles interatomic potential of silica applied to molecular dynamics*, Phys. Rev. Lett. **61** (1988), 869.

[Tsu02] Ying Y. Tsui, *EE 645 - laser matter interaction: Class notes*, Edmonton, 2002.

[vAB95] M. von Allmen and A. Blatter, *Laser-beam interactions with materials: Physical principles and applications*, 2nd ed., Springer-Verlag, Berlin, 1995.

[vBKvS90] B.W.H. van Beest, G.J. Kramer, and R.A. van Santen, *Force fields for silicas and aluminophosphates based on ab initio calculations*, Phys. Rev. Lett. **64** (1990), 1955.

[VC00] S.G. Volz and G. Chen, *Molecular-dynamics simulation of thermal conductivity of silicon crystals*, Phys. Rev. B **61** (2000), 2651.

[VTS79] J.A. Van Vechten, R. Tsu, and F.W. Saris, *Nonthermal pulsed laser annealing of Si; plasma annealing*, Physics Letters A **74** (1979), 422.

[VTSH79] J.A. Van Vechten, R. Tsu, F.W. Saris, and D. Hoonhout, *Reasons to believe pulsed laser annealing of Si does not involve simple thermal melting*, Physics Letters A **74** (1979), 417.

[Wei77] T. Weiland, *A discretization method for the solution of maxwell's equations for six-component fields*, Arch. für Elektron. und Übertrag.tech. **31** (1977), 116.

[WIOM00] K. Watanabe, Y. Ishizaka, E. Ohmura, and I. Miyamoto, *Analysis of laser ablation process in semiconductor due to ultrashort pulsed laser with molecular dynamics simulation*, Proc. SPIE - Laser Applications in Microelectronic and Optoelectronic Manufacturing V, vol. 3933, 2000, p. 46.

[WS75] Y. Waseda and K. Suzuki, *Structure of molten silicon and germanium by x-ray diffraction (and calculation of resistivity and thermoelectric power)*, Z. Phys. B **20** (1975), 339.

[ZG99a] L.V. Zhigilei and B.J. Garrison, *Mechanisms of laser ablation from molecular dynamics simulations: dependence on the initial temperature and pulse duration*, Appl. Phys. A **69** (1999), S75.

[ZG99b] ———, *Mesoscopic breathing sphere model for computer simulation of laser ablation and damage*, Int. Conf. on Modeling and Sim. of Microsys., 1999, p. 138.

[ZG99c] ———, *Molecular dynamics simulation study of the fluence dependence of particle yield and plume composition in laser desorption and ablation of organic solids*, Appl. Phys. Lett. **74** (1999), 1341.

[ZG99d] ———, *Pressure waves in microscopic simulations of laser ablation*, Mat. Res. Soc. Proc., vol. 538, 1999, p. 491.

[ZG00] ———, *Microscopic mechanisms of laser ablation of organic solids in the thermal stress confinement irradiation regimes*, J. of Appl. Phys. **88** (2000), 1281.

[ZGB99] K. Zickfeld, M.E. Garcia, and K.H. Bennemann, *Theoretical study of the laser-induced femtosecond dynamics of small Si_n clusters*, Phys. Rev. B **59** (1999), 13422.

[ZGZ02] M.I. Zeifman, B.J. Garrison, and Leonid V. Zhigilei, *Combined molecular dynamics: Direct simulation monte carlo computational study of laser ablation plume evolution*, J. of Appl. Phys. **92** (2002), 2181.

[Zha01] Z. Zhang, *Numerical simulation of short-pulsed laser processing of materials*, Numer. Heat Transf. A **40** (2001), 497.

[ZKG97] L.V. Zhigilei, P.B.S. Kodali, and B.J. Garrison, *Molecular dynamics model for laser ablation and desorption of organic solids*, J. Phys. Chem. B **101** (1997), 2028.

[Zwa65] R. Zwanzig, *Time-correlation functions and transport coefficients in statistical mechanics*, Annu. Rev. Phys. Chem. **16** (1965), 67.

Appendix A

Stillinger-Weber Force Calculation

In this section we present the detailed calculation of the force due to the Stillinger-Weber potential.

The force $\vec{F}_i$ acting on particle i due to the interaction with all other atoms is given by

$$\vec{F}_i = -\nabla_i \phi(\vec{r}_1, \vec{r}_2, ..., \vec{r}_n) \tag{A.1}$$

$$= -\nabla_i \left\{ \sum_{\substack{j \\ (j \neq i)}} V_2(r_{ij}) + \sum_{\substack{j<k \\ (j \neq i; k \neq i)}} h(\vec{r}_i, \vec{r}_j, \vec{r}_k) \right\} \tag{A.2}$$

$$= \sum_{\substack{j \\ (j \neq i)}} \left\{ -\nabla_i V_2(r_{ij}) \right\} + \sum_{\substack{j<k \\ (j \neq i; k \neq i)}} \left\{ -\nabla_i h(\vec{r}_i, \vec{r}_j, \vec{r}_k) \right\} \tag{A.3}$$

$$= \sum_{\substack{j \\ (j \neq i)}} \vec{F}_{ij} + \sum_{\substack{j<k \\ (j \neq i; k \neq i)}} \vec{F}_{ijk} \tag{A.4}$$

where $r_{ij} \equiv |\vec{r}_{ij}|$ and

$$\vec{F}_{ijk} = \vec{F}_{ijk}^{(i)} + \vec{F}_{jik}^{(i)} + \vec{F}_{kij}^{(i)} \tag{A.5}$$

$\vec{F}_{ijk}^{(n)}$ is the force acting on particle n due to the interaction of particle i with its neighbouring particles j and k.

For $r_{ij} < a$, the 2-body part of the force is:

$$\vec{F}_{ij} = -\nabla_i V_2(r_{ij}) \tag{A.6}$$

$$= -\frac{\partial}{\partial r}\left\{(A_1 r^{-4} - A_2)\exp\left(\frac{\sigma}{r-a}\right)\right\}\hat{r}_{ij} \tag{A.7}$$

$$= -\left\{(A_1 r^{-4} - A_2)\left[\frac{\partial}{\partial r}e^{\left(\frac{\sigma}{r-a}\right)}\right] + \left[\frac{\partial}{\partial r}(A_1 r^{-4} - A_2)\right]e^{\left(\frac{\sigma}{r-a}\right)}\right\}\hat{r}_{ij} \tag{A.8}$$

$$= -\left\{(A_1 r^{-4} - A_2)\left[\frac{-\sigma}{(r-a)^2}e^{\left(\frac{\sigma}{r-a}\right)}\right] + \left[(-4A_1 r^{-5})\right]e^{\left(\frac{\sigma}{r-a}\right)}\right\}\hat{r}_{ij} \tag{A.9}$$

$$= \left[(A_1 r^{-4} - A_2)\frac{\sigma}{(r-a)^2} + 4A_1 r^{-5}\right]\exp\left(\frac{\sigma}{r-a}\right)\hat{r}_{ij} \tag{A.10}$$

(A.11)

The force acting on particle i due its neighbours j and k is given as follows:

$$\vec{F}_{ijk}^{(i)} = -\nabla_i h(\vec{r}_i, \vec{r}_j, \vec{r}_k) \tag{A.12}$$

$$= -\nabla_i\left\{\lambda\exp\left(\frac{\gamma\sigma}{r_{ij}-a}\right)\exp\left(\frac{\gamma\sigma}{r_{ik}-a}\right)\times(\cos\theta_{jik} + \frac{1}{3})^2\right\} \tag{A.13}$$

$$= -\lambda\,\nabla_i\left\{u_{ij}\,u_{ik}\,\omega_{ijk}^2\right\} \tag{A.14}$$

$$= -\lambda\left\{\nabla_i u_{ij}\,u_{ik}\,\omega_{ijk}^2 + u_{ij}\,\nabla_i u_{ik}\,\omega_{ijk}^2 + u_{ij}\,u_{ik}\,\nabla_i\omega_{ijk}^2\right\} \tag{A.15}$$

(A.16)

with

$$u_{ij} \equiv u(\vec{r}_{ij}) = \exp\left(\frac{\gamma\sigma}{r_{ij}-a}\right) \tag{A.17}$$

$$u_{ik} \equiv u(\vec{r}_{ik}) = \exp\left(\frac{\gamma\sigma}{r_{ik}-a}\right) \tag{A.18}$$

$$\omega_{ijk} \equiv \omega(\vec{r}_{ij},\vec{r}_{ik}) = \cos\theta_{jik} + \frac{1}{3} = \hat{r}_{ij}\cdot\hat{r}_{ik} + \frac{1}{3} \tag{A.19}$$

(A.20)

The gradient of u_{ij} can be calculated as follows:

$$\nabla_i u_{ij} = \nabla_i \exp\left(\frac{\gamma\sigma}{r_{ij}-a}\right) = \nabla_i\left(\frac{\gamma\sigma}{r_{ij}-a}\right) u_{ij} \tag{A.21}$$

$$= -\frac{\gamma\sigma}{(r_{ij}-a)^2}\nabla_i\left(|\vec{r}_j-\vec{r}_i|-a\right) u_{ij} \tag{A.22}$$

$$= \frac{\gamma\sigma}{(r_{ij}-a)^2} u_{ij}\,\hat{r}_{ij} \tag{A.23}$$

Since

$$\nabla_i\left(|\vec{r}_j-\vec{r}_i|-a\right) = \hat{x}\frac{\partial}{\partial x_i}\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2}+ \tag{A.24}$$

$$\hat{y}\frac{\partial}{\partial y_i}\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2}+ \tag{A.25}$$

$$\hat{z}\frac{\partial}{\partial z_i}\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2} \tag{A.26}$$

$$= \frac{-2(x_j-x_i)\hat{x}-2(y_j-y_i)\hat{y}-2(z_j-z_i)\hat{z}}{2\sqrt{(x_j-x_i)^2+(y_j-y_i)^2+(z_j-z_i)^2}} \tag{A.27}$$

$$= \frac{-\vec{r}_{ij}}{r_{ij}} = -\hat{r}_{ij} \tag{A.28}$$

Similar for $\nabla_i u_{ik}$. For the gradient of the angular part we have:

$$\nabla_i \omega_{ijk} = \nabla_i \left(\hat{r}_{ij} \cdot \hat{r}_{ik} + \frac{1}{3} \right) = \nabla_i \left(\frac{\vec{r}_{ij} \cdot \vec{r}_{ik}}{r_{ij} r_{ik}} \right) \tag{A.29}$$

$$= \frac{1}{(r_{ij} r_{ik})^2} \left[\nabla_i (\vec{r}_{ij} \cdot \vec{r}_{ik})\, r_{ij} r_{ik} - \nabla_i (r_{ij} r_{ik})(\vec{r}_{ij} \cdot \vec{r}_{ik}) \right] \tag{A.30}$$

$$\nabla_i (r_{ij} r_{ik}) = (\nabla_i r_{ij})\, r_{ik} + r_{ij}\, (\nabla_i r_{ik}) = -r_{ik} \hat{r}_{ij} - r_{ij} \hat{r}_{ik} \tag{A.31}$$

$$\nabla_i (\vec{r}_{ij} \cdot \vec{r}_{ik}) = \nabla_i (x_{ij} x_{ik} + y_{ij} y_{ik} + z_{ij} z_{ik}) \tag{A.32}$$

$$= \frac{\partial}{\partial x_i} (x_{ij} x_{ik}) \hat{x} + \frac{\partial}{\partial y_i} (y_{ij} y_{ik}) \hat{y} + \frac{\partial}{\partial z_i} (z_{ij} z_{ik}) \hat{z} \tag{A.33}$$

$$= \left[\frac{\partial}{\partial x_i} (x_j - x_i) x_{ik} + x_{ij} \frac{\partial}{\partial x_i} (x_k - x_i) \right] \hat{x} + \ldots \tag{A.34}$$

$$= (-x_{ik} - x_{ij}) \hat{x} + (-y_{ik} - y_{ij}) \hat{y} + (-z_{ik} - z_{ij}) \hat{z} \tag{A.35}$$

$$= -\vec{r}_{ij} - \vec{r}_{ik} \tag{A.36}$$

Now substituting equations A.31 and A.36 into equation A.30 we get:

$$\nabla_i \omega_{ijk} = \frac{1}{(r_{ij} r_{ik})^2} \left[(-\vec{r}_{ij} - \vec{r}_{ik})\, r_{ij} r_{ik} - (-r_{ik} \hat{r}_{ij} - r_{ij} \hat{r}_{ik})(\vec{r}_{ij} \cdot \vec{r}_{ik}) \right] \tag{A.37}$$

$$= \frac{1}{(r_{ij} r_{ik})^2} \left[\frac{\vec{r}_{ij} \cdot \vec{r}_{ik}}{r_{ij} r_{ik}} \frac{r_{ik}}{r_{ij}} \vec{r}_{ij} - \vec{r}_{ij} + \frac{\vec{r}_{ik} \cdot \vec{r}_{ij}}{r_{ik} r_{ij}} \frac{r_{ij}}{r_{ik}} \vec{r}_{ik} - \vec{r}_{ik} \right] \tag{A.38}$$

$$= \frac{1}{(r_{ij} r_{ik})^2} \left[\left(\frac{\vec{r}_{ij} \cdot \vec{r}_{ik}}{r_{ij}^2} - 1 \right) \vec{r}_{ij} + \left(\frac{\vec{r}_{ik} \cdot \vec{r}_{ij}}{r_{ik}^2} - 1 \right) \vec{r}_{ik} + \right] \tag{A.39}$$

Now we can insert equations A.17, A.18, A.23, and A.39 into equation A.15.

The force acting on particles j and k due to this interaction is given as follows:

$$\vec{F}_{ijk}^{(j)} = +\nabla_j h(\vec{r}_i,\vec{r}_j,\vec{r}_k) \tag{A.40}$$

$$= \lambda \left\{\nabla_j u_{ij}\, u_{ik}\, \omega_{ijk}^2 + u_{ij}\, u_{ik}\, 2\omega_{ijk} \nabla_{ij}\omega_{ijk}\right\} \tag{A.41}$$

The gradient of the angular component is obtained in a similar fashion as before, except now $\vec{r}_{ik}$ is held constant:

$$\nabla_{ij}\omega_{ijk} = \nabla_{ij}\left(\hat{r}_{ij}\cdot\hat{r}_{ik}+\frac{1}{3}\right) = \nabla_{ij}\left(\frac{\vec{r}_{ij}\cdot\vec{r}_{ik}}{r_{ij}r_{ik}}\right) \tag{A.42}$$

$$= \frac{1}{(r_{ij}r_{ik})^2}\left[\nabla_{ij}(\vec{r}_{ij}\cdot\vec{r}_{ik})\, r_{ij}r_{ik} - (\nabla_i r_{ij})\, r_{ik}(\vec{r}_{ij}\cdot\vec{r}_{ik})\right] \tag{A.43}$$

$$= \frac{1}{r_{ij}r_{ik}}\left[-\vec{r}_{ik}+\frac{r_{ik}}{r_{ij}r_{ik}}\hat{r}_{ij}(\vec{r}_{ij}\cdot\vec{r}_{ik})\right] \tag{A.44}$$

$$= \frac{1}{r_{ij}r_{ik}}\left[\frac{\vec{r}_{ij}\cdot\vec{r}_{ik}}{r_{ij}^2}\vec{r}_{ij}-\vec{r}_{ik}\right] \tag{A.45}$$

Inserting eqn. A.45 into A.41 we get:

$$\vec{F}_{ijk}^{(j)} = \lambda\left\{(\nabla_i u_{ij})\, u_{ik}\, \omega_{ijk}^2 + \frac{2\omega_{ijk}\, u_{ij}\, u_{ik}}{r_{ij}r_{ik}}\left[\frac{\vec{r}_{ij}\cdot\vec{r}_{ik}}{r_{ij}^2}\vec{r}_{ij}-\vec{r}_{ik}\right]\right\} \tag{A.46}$$

$$\vec{F}_{ijk}^{(k)} = \lambda\left\{u_{ij}\,(\nabla_i u_{ik})\, \omega_{ijk}^2 + \frac{2\omega_{ijk}\, u_{ij}\, u_{ik}}{r_{ij}r_{ik}}\left[\frac{\vec{r}_{ij}\cdot\vec{r}_{ik}}{r_{ik}^2}\vec{r}_{ik}-\vec{r}_{ij}\right]\right\} \tag{A.47}$$

We can perform a simple check to see that we didn't make a mistake: We know that the forces must add up to zero, so

$$\vec{F}_{ijk}^{(i)} + \vec{F}_{ijk}^{(j)} + \vec{F}_{ijk}^{(k)} = 0 \tag{A.48}$$

and indeed we find that this is the case.

Appendix B

Thermal Conductivity from MD Simulation

The heat flow equation for a 1-D system (or a 3-D system that is infinite in two dimensions) is:

$$\rho c_V \frac{\partial T}{\partial t} = \frac{\partial}{\partial z}\left(\kappa \frac{\partial T}{\partial z}\right) \tag{B.1}$$

where ρ is the density, c_V is the specific heat (at constant volume), κ is the thermal conductivity, and $T = T(z,t)$ is the temperature. z and t are the independent variables in space and time, respectively. Let us have constant boundary conditions:

$$\begin{aligned} T(0,t) &= T_0 \\ T(L,t) &= T_0 \\ T(x,0) &= T_0 + f(x) \end{aligned} \tag{B.2}$$

To simplify the following derivation, we shall choose $T_0 = 0$. Later it will be trivial to set T_0 to an arbitrary value and adjust the solution accordingly. If the heat conductivity is constant in space (and time), we can write:

$$\frac{1}{\alpha^2}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial z^2} \tag{B.3}$$

where the quantity α^2 is the thermal diffusivity and is defined by:

$$\alpha^2 = \frac{\kappa}{\rho c_V} \tag{B.4}$$

Assuming that the temperature can be written as a product of a function of z only and a function of t only, we write:

$$T(z,t) = T_z(z)\,T_t(t) \tag{B.5}$$

substituting this into equation B.3 we get:

$$\frac{T_x''}{T_x} = \frac{1}{\alpha^2}\frac{T_t'}{T_t} = -\sigma \tag{B.6}$$

where the primes indicate ordinary differentiation with respect to the independent variable (x or t). Since equation B.6 must be valid for $0 < x < L$ and $t > 0$, σ is a constant (it turns out that, given the boundary conditions, $\sigma > 0$ [BD97]). Thus we can separate equation B.6 into a system of equations

as follows:

$$T_x'' + \sigma T_x = 0 \tag{B.7}$$

$$T_t' + \alpha^2 \sigma T_t = 0 \tag{B.8}$$

The general solution to eq. B.7 and B.8 are

$$T_x(x) = k_1 \cos(\alpha^2 \sigma x) + k_2 \sin(\alpha^2 \sigma x) \tag{B.9}$$

$$T_t(t) = C \exp(-\alpha^2 \sigma t) \tag{B.10}$$

Where k_1, k_2, and C_t are constants. Substituting the boundary conditions, we find that

$$k_1 = 0 \tag{B.11}$$

$$\sigma = (n\pi/L)^2, \quad n = 1,2,3,... \tag{B.12}$$

Now multiplying equations B.9 and B.10 gives:

$$T_n(x,t) = C_n \sin\left(\frac{n\pi z}{L}\right) \exp\left(-\frac{n^2\pi^2\alpha^2 t}{L^2}\right) \tag{B.13}$$

The differential equation B.3 and the boundary conditions (eqns. B.2) are linear and homogeneous, thus by the principle of superposition, we find that:

$$T(x,t) = T_0 + \sum_{n=1}^{\infty} T_n(x,t) = T_0 + \sum_{n=1}^{\infty} C_n \sin\left(\frac{n\pi z}{L}\right) \exp\left(-\frac{n^2\pi^2\kappa t}{\rho c_V L^2}\right) \tag{B.14}$$

The coefficients C_n are determined from the boundary conditions.

Now we shall choose a convenient initial temperature profile to determine the thermal conductivity κ. For this we use

$$T(x,0) = \Delta T \sin\left(\frac{\pi z}{L}\right) \tag{B.15}$$

such that the coefficients C_n are zero for all n with the exception of C_1, which is non-zero and denoted as ΔT. The temporal evolution of the profile is then given by:

$$T(x,t) = T_0 + \Delta T \sin\left(\frac{\pi z}{L}\right) \exp\left(-\frac{\pi^2\kappa t}{\rho c_V L^2}\right) \tag{B.16}$$

Appendix C

Figures

C.1 Removed particles over time

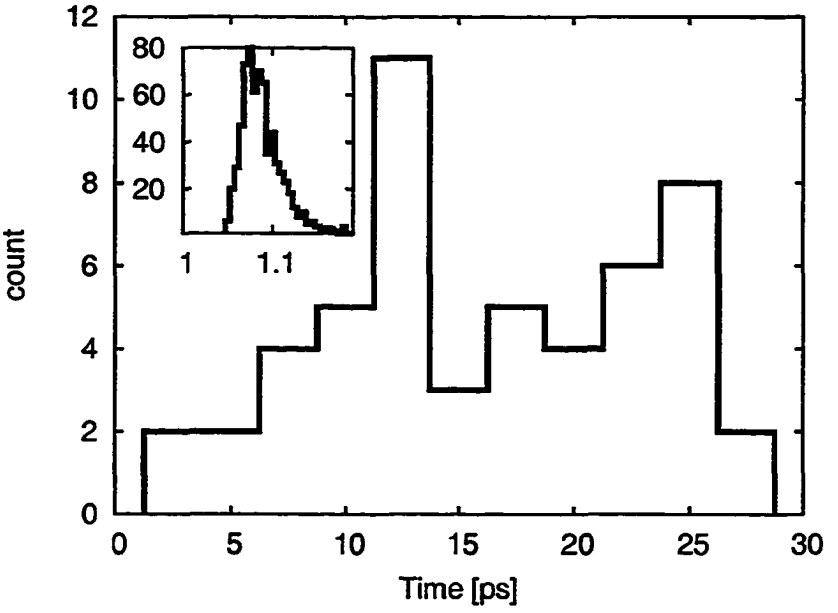


Figure C.1: Count of ablated atoms and electrons (inset) for 50 fs laser pulse (1/e) with a fluence of 0.10 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.05$ ps

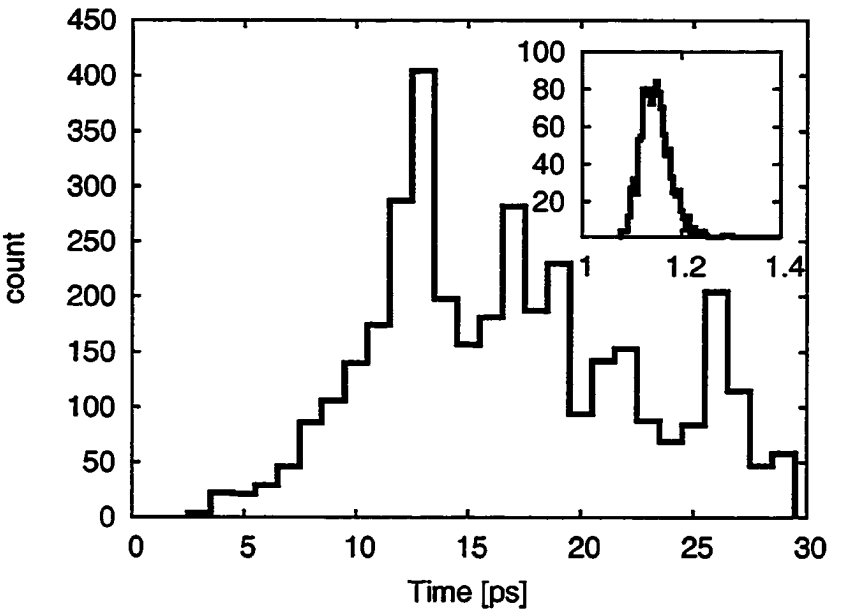


Figure C.2: Count of ablated atoms and electrons (inset) for 100 fs laser pulse (1/e) with a fluence of 0.16 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.1$ ps

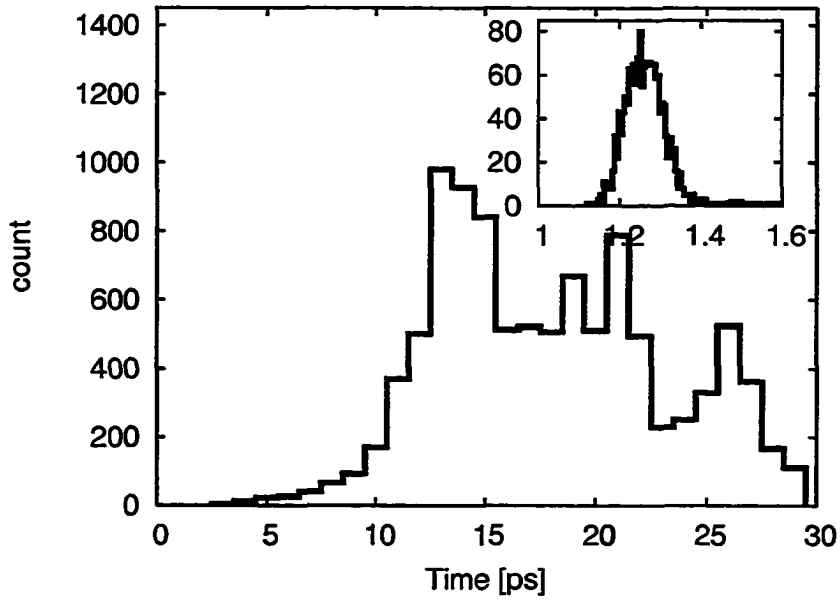


Figure C.3: Count of ablated atoms and electrons (inset) for 200 fs laser pulse (1/e) with a fluence of 0.24 J/cm^2 ($\lambda = 800$nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.2$ps

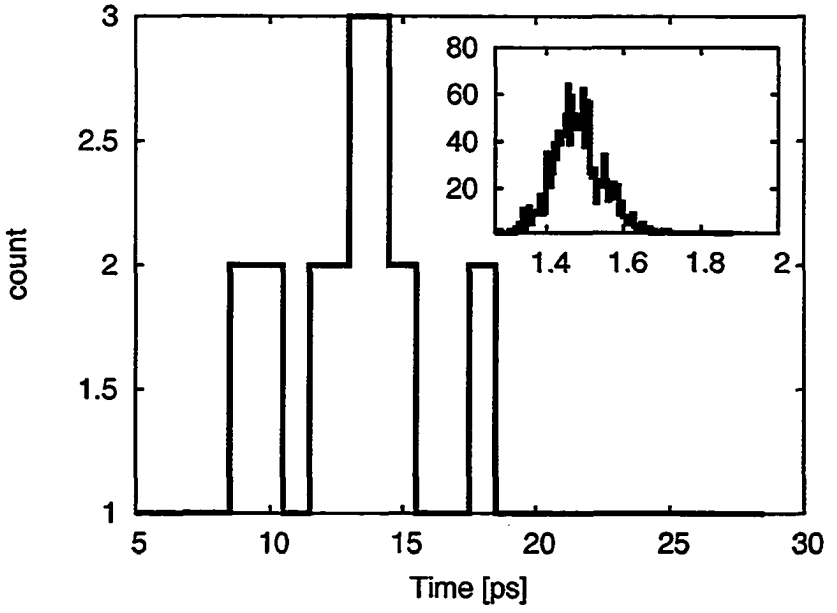


Figure C.4: Count of ablated atoms and electrons (inset) for 400 fs laser pulse (1/e) with a fluence of 0.30 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.4$ ps

C.2 Ablation sequences

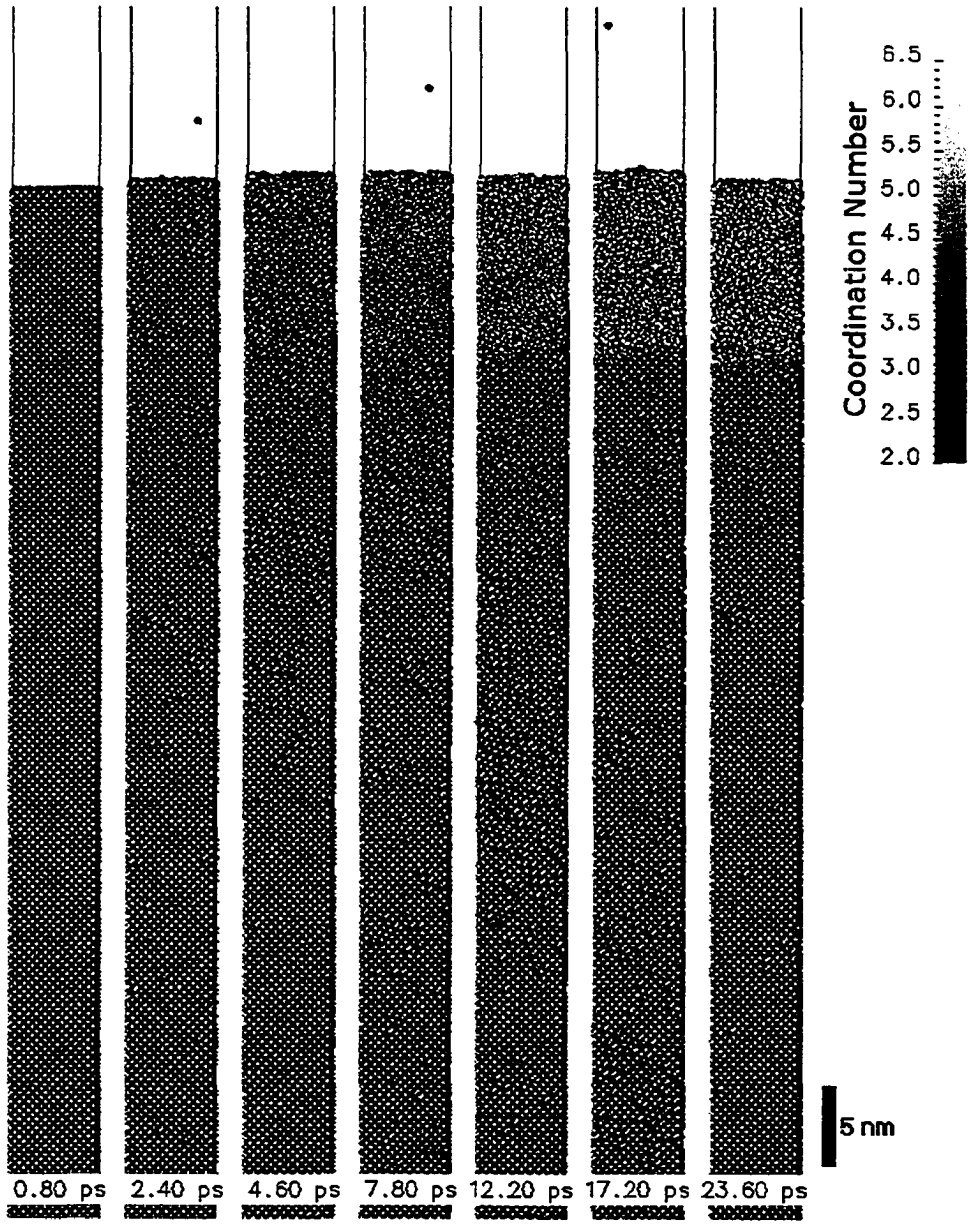


Figure C.5: Ablation sequence for 50 fs laser pulse (1/e) with a fluence of 0.08 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.05$ ps

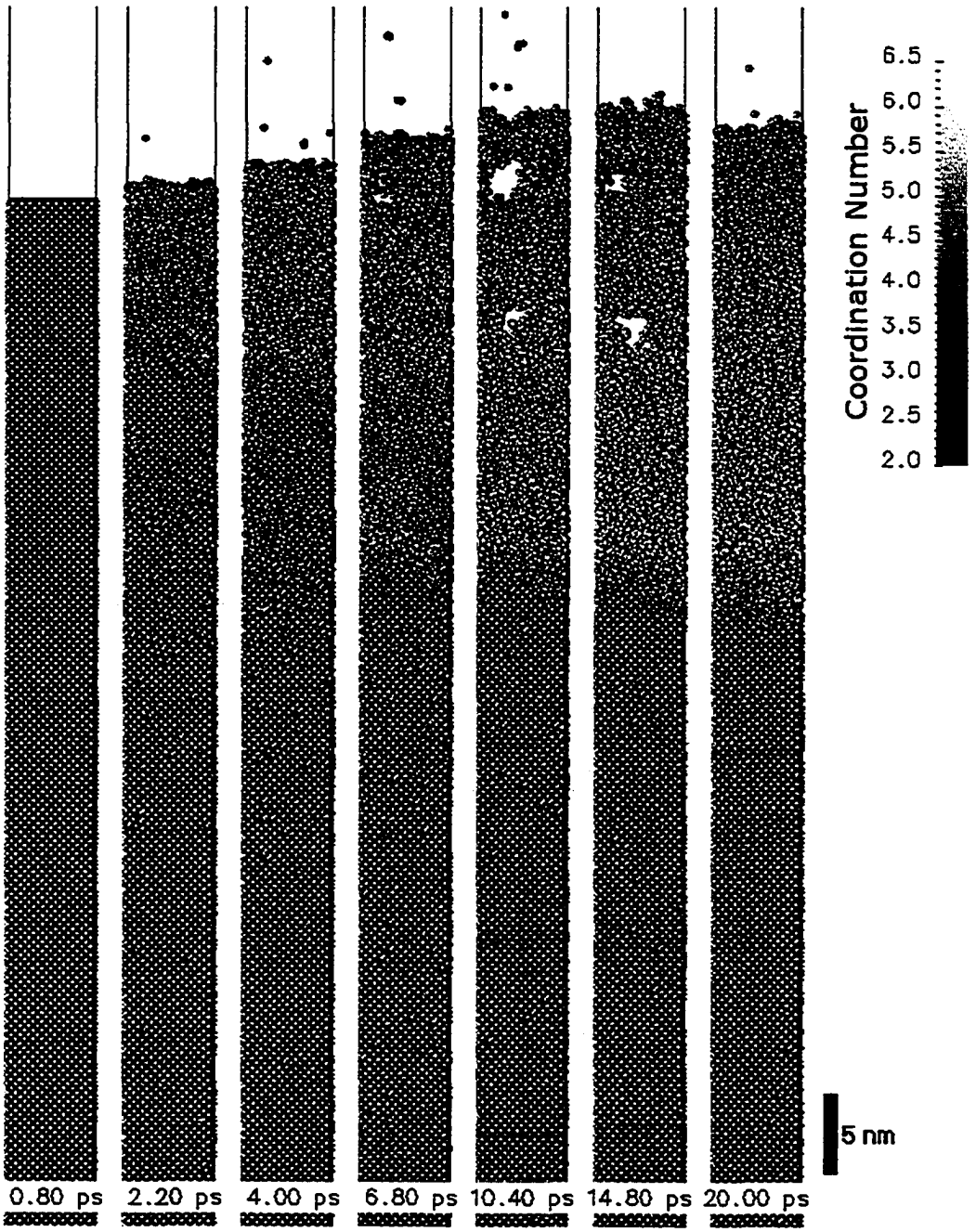


Figure C.6: Ablation sequence for 50 fs laser pulse (1/e) with a fluence of 0.10 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.05$ ps

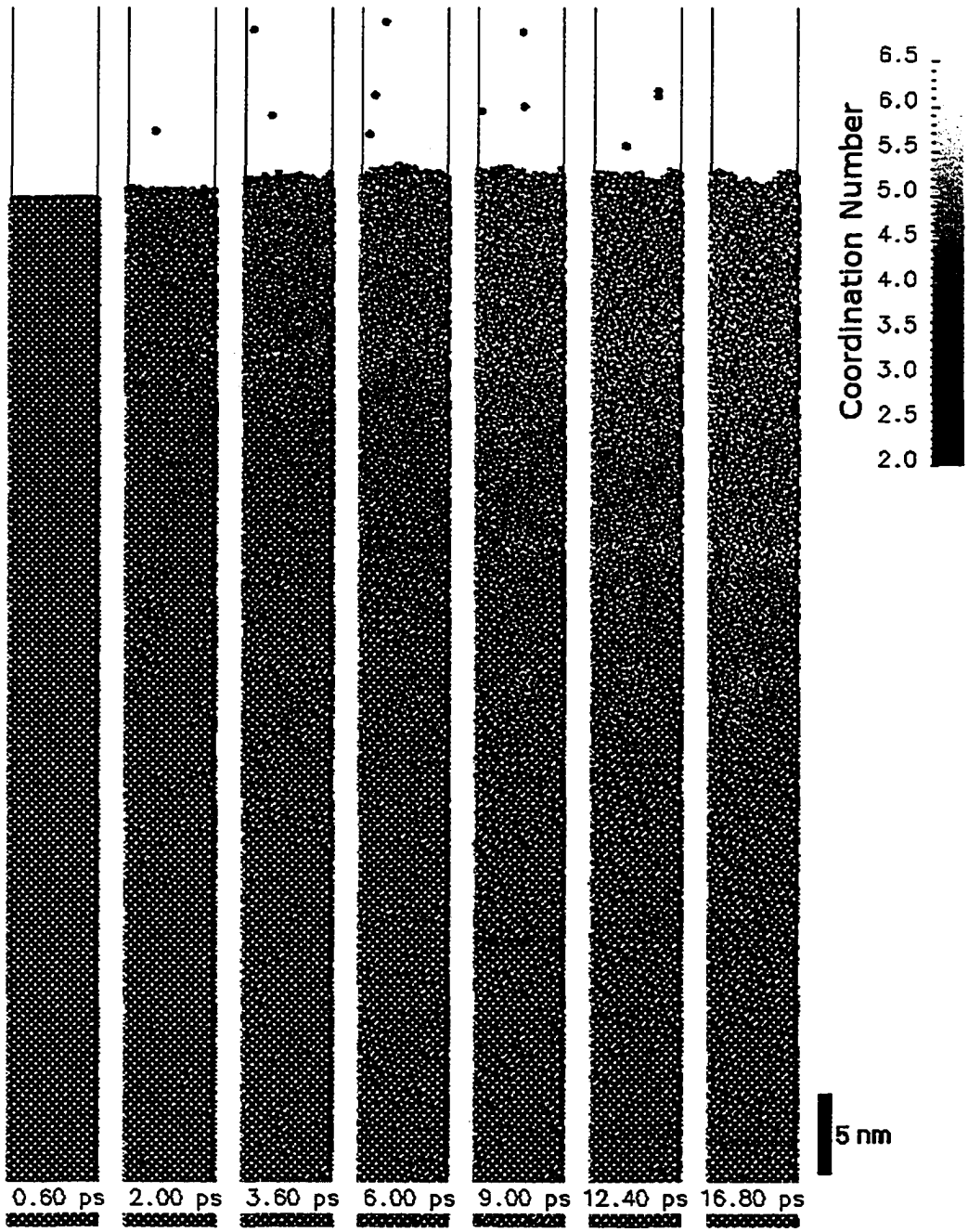


Figure C.7: Ablation sequence for 100 fs laser pulse (1/e) with a fluence of 0.13 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.1$ ps

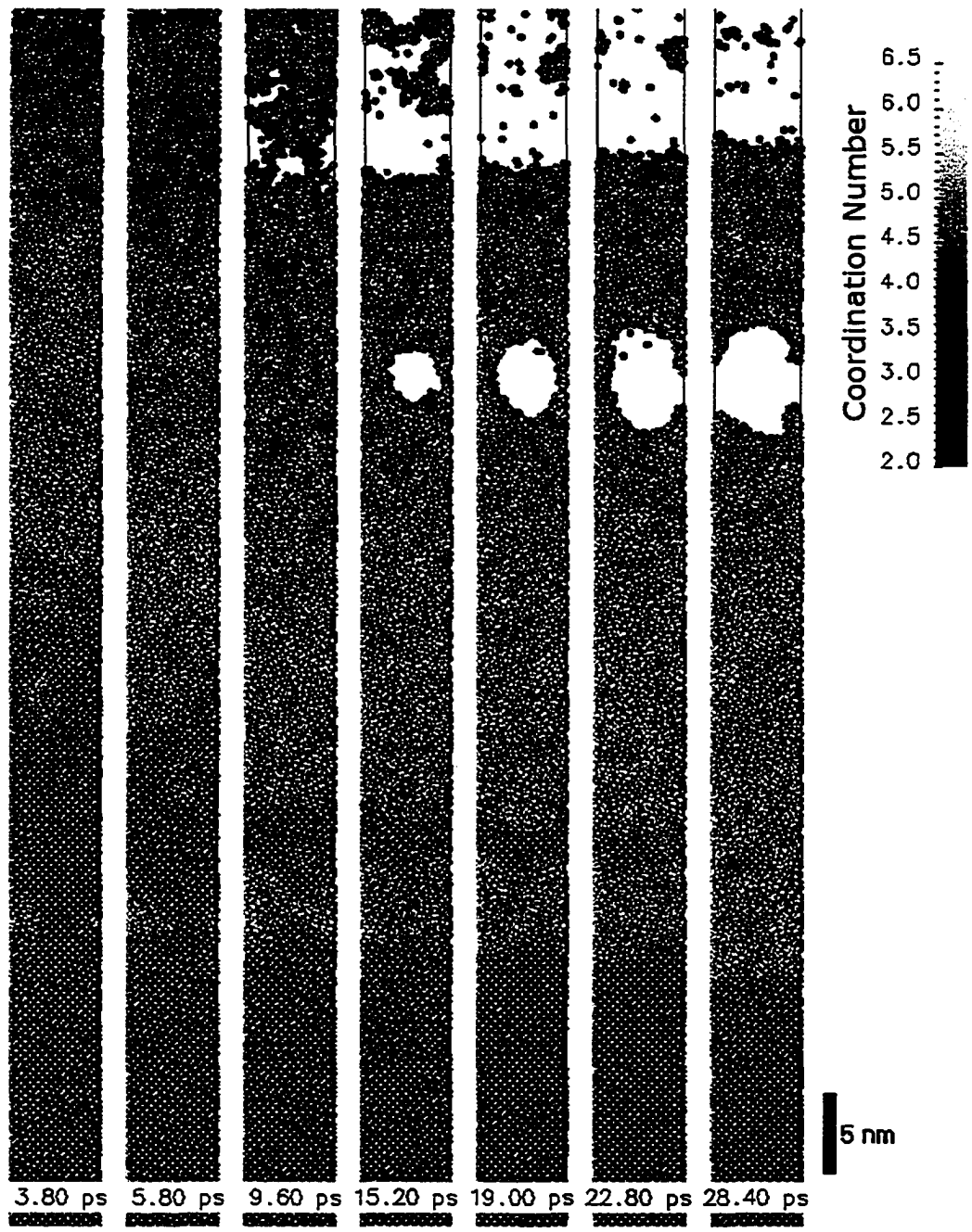


Figure C.8: Ablation sequence for 100 fs laser pulse (1/e) with a fluence of 0.16 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.1$ ps

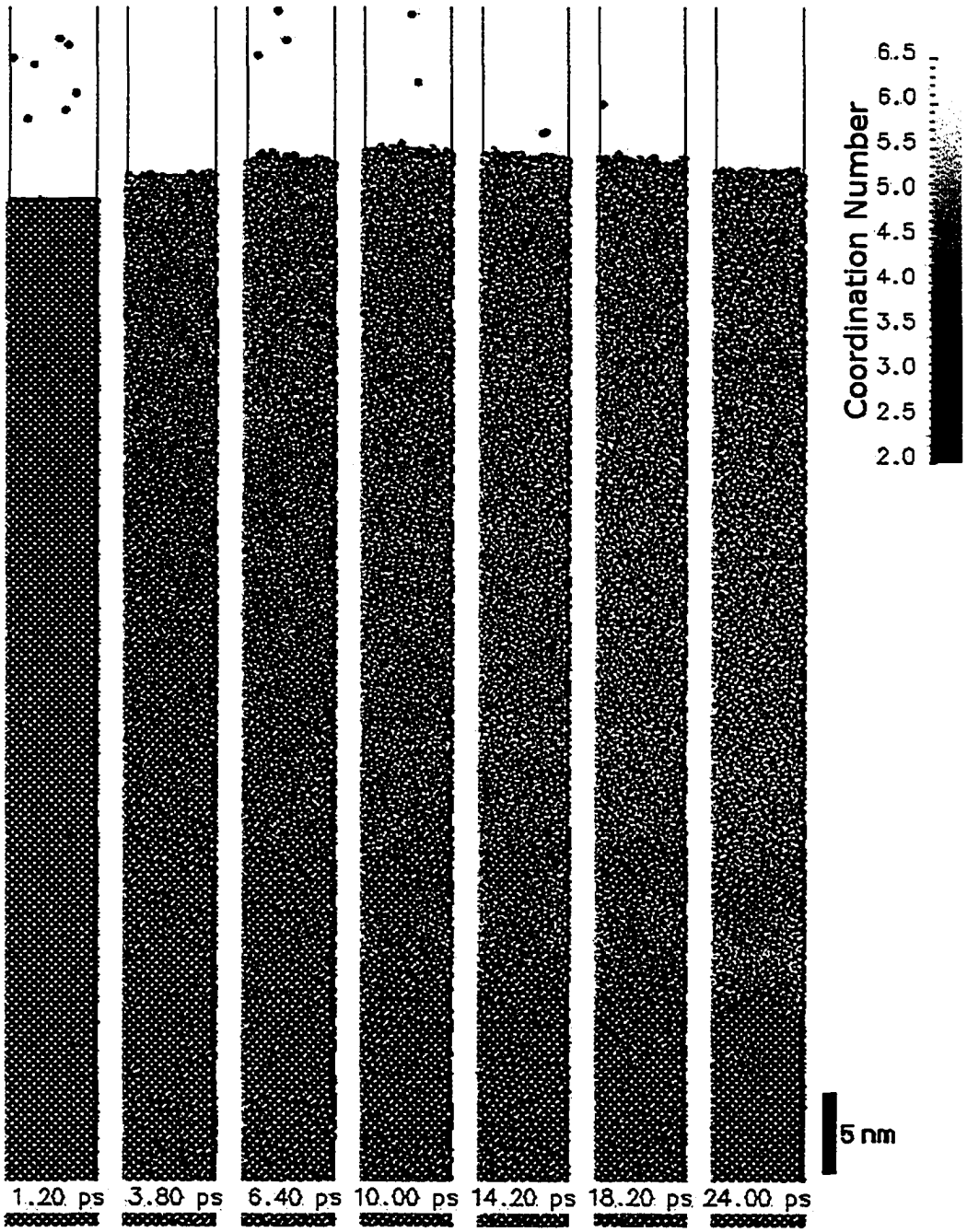


Figure C.9: Ablation sequence for 200 fs laser pulse (1/e) with a fluence of 0.20 J/cm^2 ($\lambda = 800$ nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.2$ ps

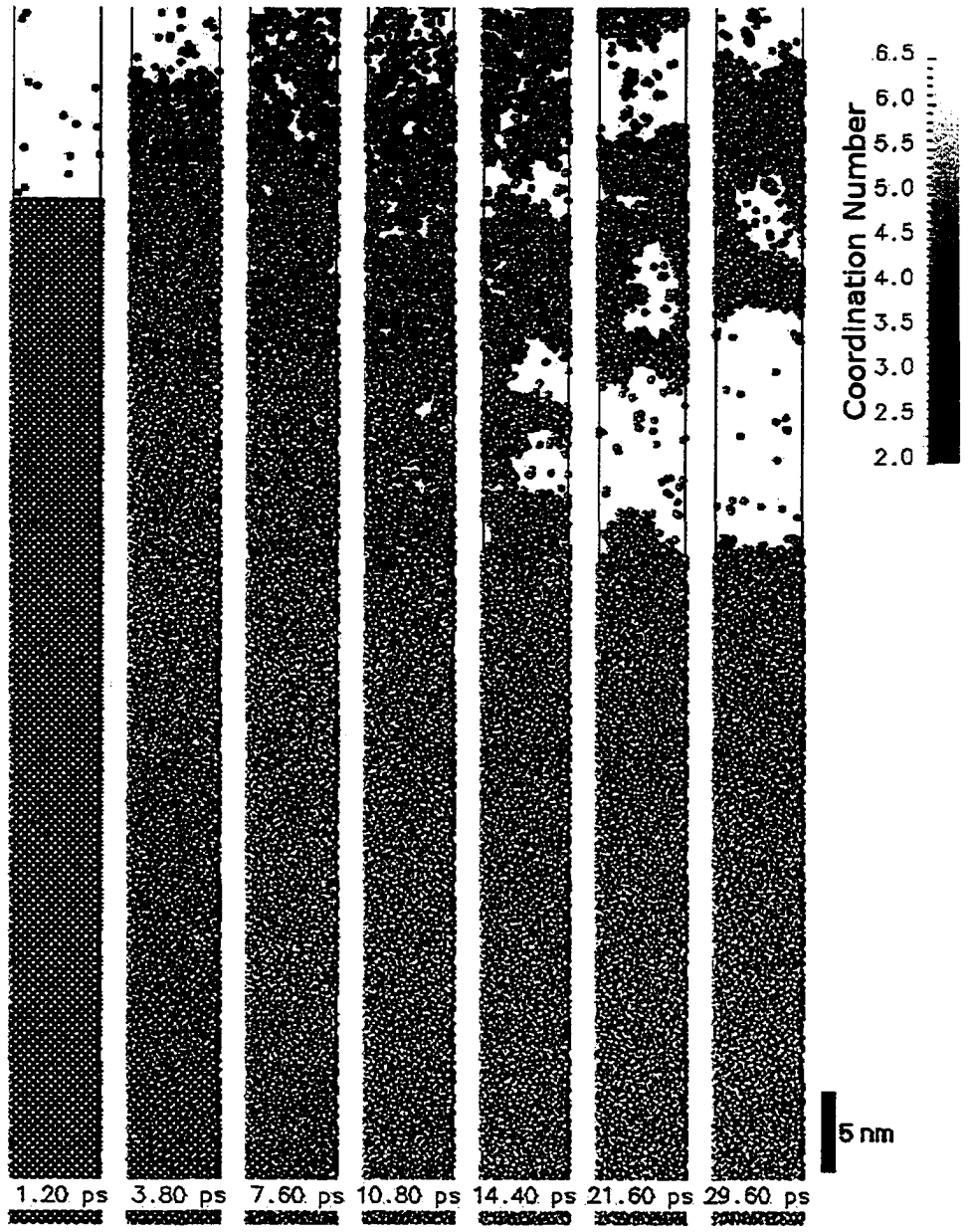


Figure C.10: Ablation sequence for 200 fs laser pulse (1/e) with a fluence of 0.24 J/cm^2 (λ = 800nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.2$ps

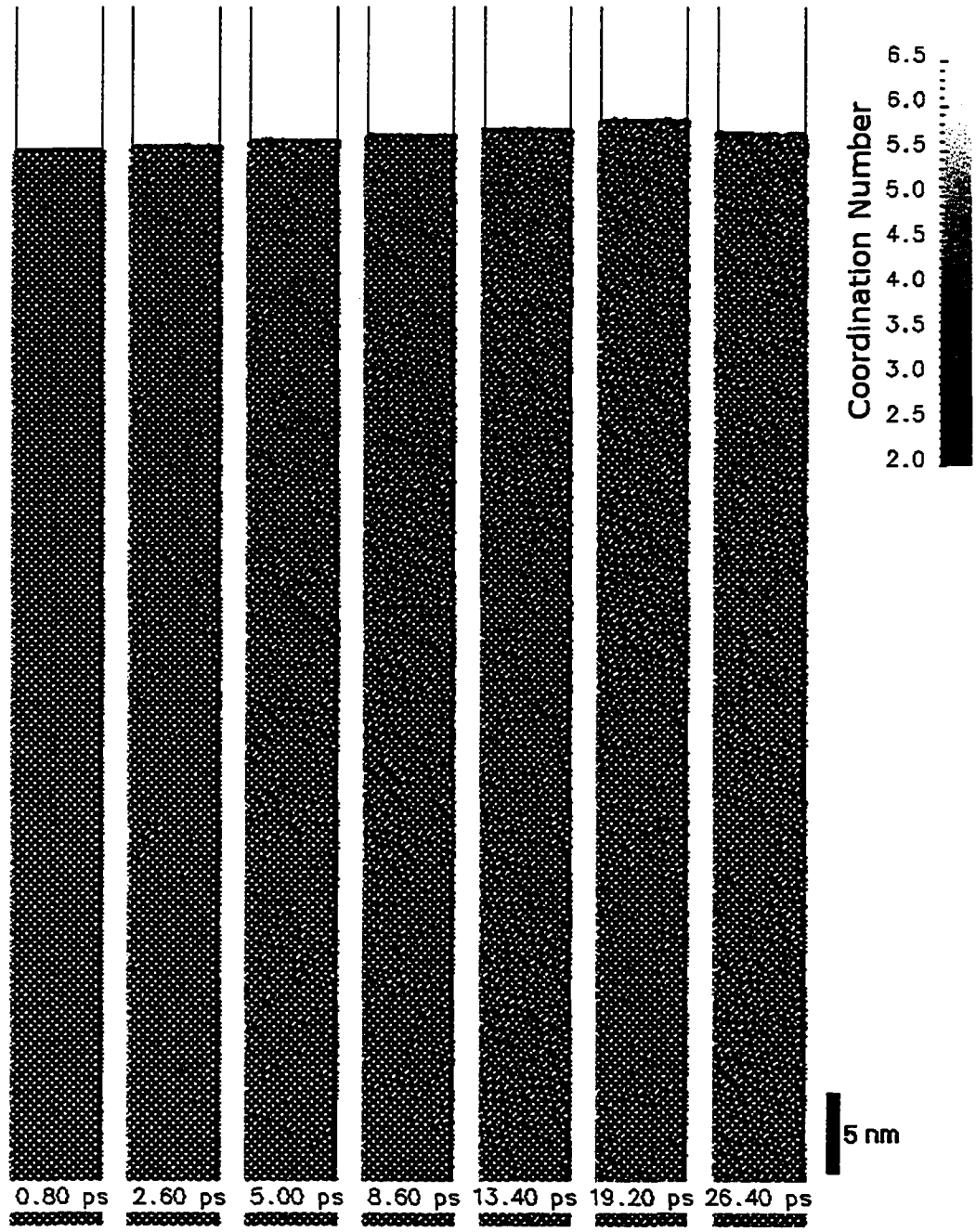


Figure C.11: Ablation sequence for 400 fs laser pulse (1/e) with a fluence of 0.22 J/cm^2 (λ = 800 nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.4$ ps

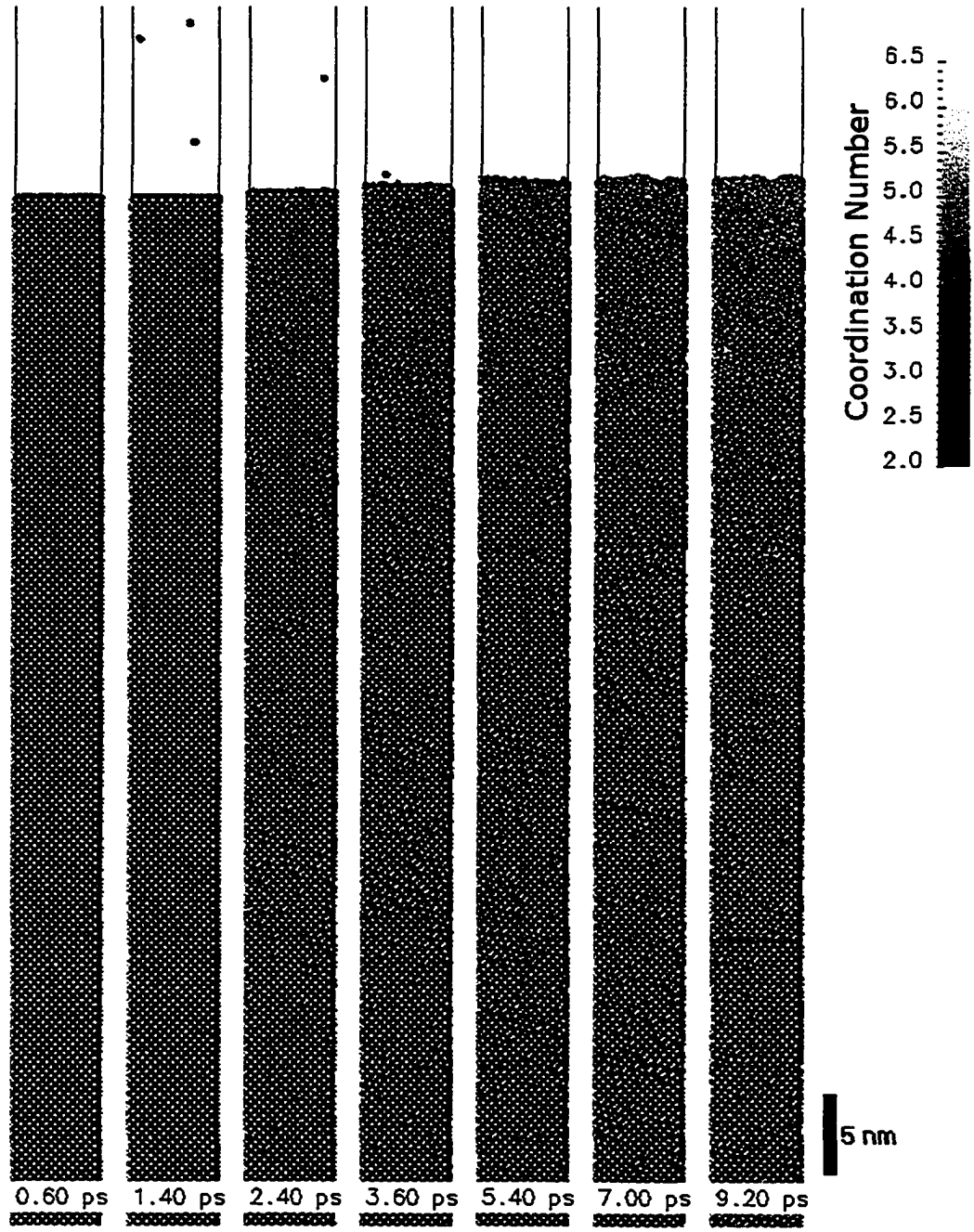


Figure C.12: Ablation sequence for 400 fs laser pulse (1/e) with a fluence of 0.26 J/cm^2 (λ = 800 nm). The laser pulse starts at t = 1 ps with peak intensity at t = 1.4 ps

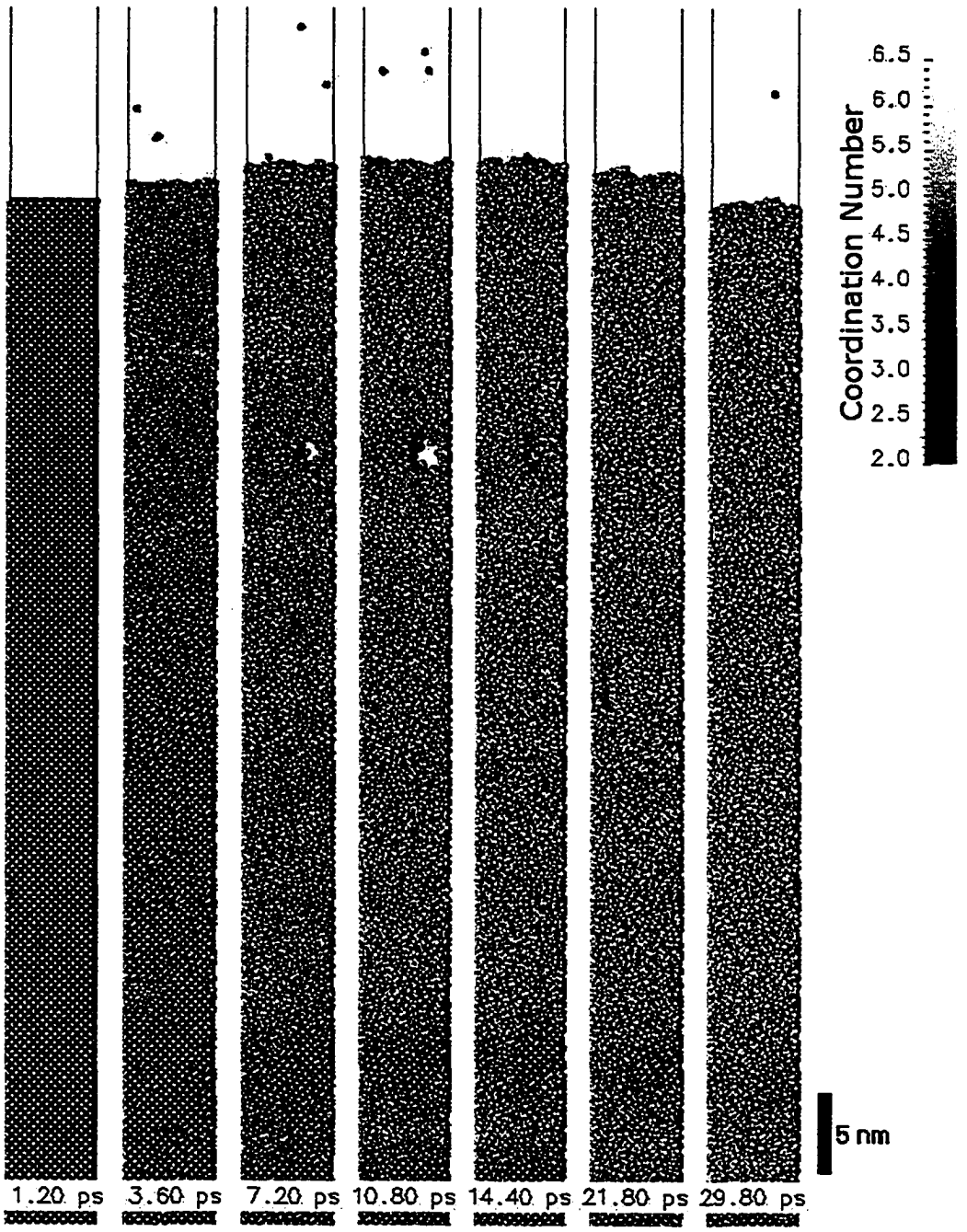


Figure C.13: Ablation sequence for 400 fs laser pulse (1/e) with a fluence of 0.30 J/cm^2 (λ = 800 nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.4$ ps

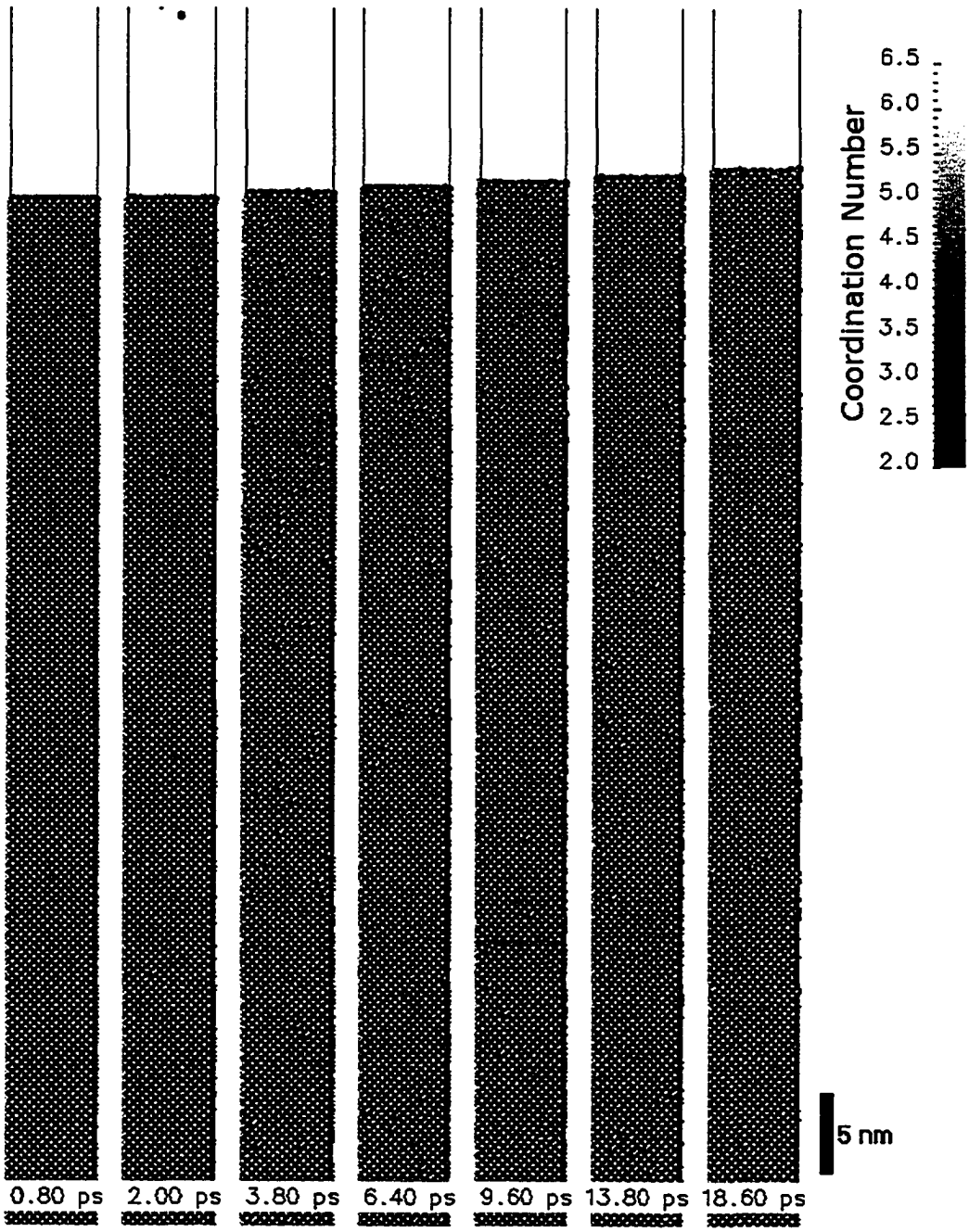


Figure C.14: Ablation sequence for 800 fs laser pulse (1/e) with a fluence of 0.36 J/cm^2 (λ = 800 nm). The laser pulse starts at t = 1 ps with peak intensity at t = 1.8 ps

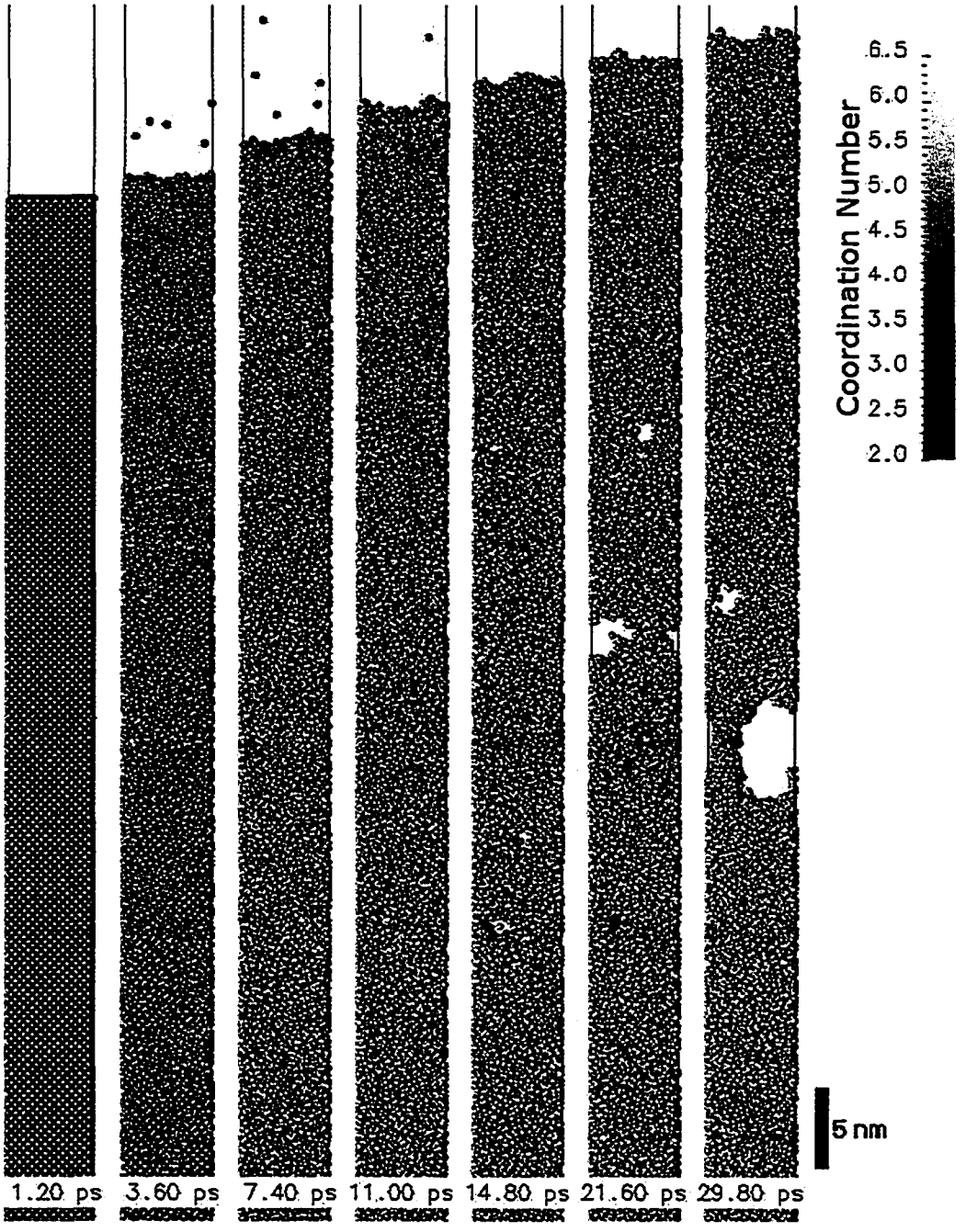


Figure C.15: Ablation sequence for 800 fs laser pulse (1/e) with a fluence of 0.50 J/cm^2 (λ = 800nm). The laser pulse starts at $t = 1$ ps with peak intensity at $t = 1.8$ ps

Appendix D

Simulation Code

D.1 Parameter File

The parameters for a given simulation run are specified in a file. The syntax is "⟨variable⟩ = ⟨value⟩", with at most one such definition per line. The order of the variables is not important. The pound sign (#) is used for comments, the remainder of the line following a pound sign is ignored.

parameters

```
#
# Simulation Parameters
#

#   *** general simulation parameters ****
TEMPERATURE     = 300          # starting bulk temperature (K)
TIMESTEP        = 1          # length of time step (fs)
TOTALTIME       = 5000         # maximum time (fs)
lastEquilibTime = 0          # no equilibration after this time (fs)
XSTP            = 0.3          # atoms: maximum change in position per step (A)
EXSTP           = 1          # electrons: maximum change in position per step (A)
speedup_1_t       = 20001          # t: change timestep to dt at time t
speedup_1_dt       = 2           # dt: change timestep to dt at time t
speedup_2_t       = 22000          # t: change timestep to dt at time t
speedup_2_dt       = 3           # dt: change timestep to dt at time t
```

```
15 VARIABLE_TIMESTEP = 0          # 1=dynamically adjust timesteps, 0=const dt

16 #   *** laser parameters ***
17 LWAVLEN      = 760          # Laser wavelength (nm)
18 LPULSLEN     = 200          # pulse length (1/e) (fs)
19 LEngDen      = 0.10          # laser pulse energy density (J/cm2)
20 LSSHAPE      = 0           # spatial shape (0=uniform, 1=gaussian)
21 LFOCUSDIA    = 25          # laser focus diameter (1/e Gauss) (A)
22 LAS_ON       = 100          # laser on after t (fs)

23 #   *** Silicon parameters ***
24 MASS         = 28.0855          # mass (amu)
25 SPACE        = 5.43095          # length of unit cell at 0K (at 300K: 5.430949) (A)
26 LATTICE_TYPE = DIA100          # lattice structure (dia100=diamond with 100 surface)
27 LATTICE_FILE = atompos.xyz          # lattice structure (dia100=diamond with 100 surface)
28 DEBYE        = 645          # debye temperature (K)
29 OVER_WEIGH   = 1          # adjust absorption to size of target
30 NUMXYM       = 5          # number of atom layers in X and Y direction
31 NUMZM        = 50          # number of atom layers in Z direction
32 FIXED_GROUND = 0          # 0 = no; 1 = yes
33 DIST_FIX     = 0          # bottom layer in which atoms are fixed (A)
34 DIST_DMP     = 50          # thickness of upper rim where atoms are damped (A)
35 INCL_HF      = 1          # couple to 1-D heatflow model (requires DIST_DMP>0)
36 SiLIFET      = 1000          # lifetime of excited "state" (1/e) (fs)
37 SiIONPOT     = 4.85          # work function for silicon (eV)
38 IONRECOM     = 500          # Ion recombination time (1/e) (fs)
39 ABS_COEF     = 3500          # linear absorption coefficient (1/cm)
40 ABS_COEFtwo  = 55          # two-photon absorption coefficient (cm/GW)
41 T_ELEC       = 6000          # average electron temp. (Kelvin)
42 EL_STEP      = 50          # number of electron-timesteps per atom-timestep
43 INIT_SPEED_ELEC = 25          # initial speed of electrons (A/fs)

44 #   *** Potential parameters ***
45 CLRC               = 20          # cut off length (A) for Coulomb potential
46 incl_Coulomb       = 0          # include Coulomb potential (1=yes,0=no)
47 incl_SW            = 1          # include Stillinger-Weber potential (1=yes,0=no)
48 SW_inclBreakBonds  = 0          # include breaking bonds (1=yes, 0=no)
49 incl_LJ            = 0          # include Lennard-Jones potential (1=yes,0=no)
50 LJcutoff           = 10          # cut-off length (A) for Lennard-Jones pot.
51 inclElectrons      = 0          # gen. electron from ionization process (1=yes,0=no)

52 #   *** control parameters ***
53 SAVE_AT            = 5          # frequency at which to save sim (fs)
54 SAVE_SCALE_AT      = 200000          # scale save frequency at this time (fs)
55 SAVE_SCALE_BY      = 2          # scale save frequency by this amount
56 STATS_START_AT     = 0 # (fs)
57 STATS_SAVE_AT      = 10          # interval of saving stats (fs)
58 STATS_COLLECT_TIME = 10          # time over which to collect stats (fs)
59 PRT_INFO           = 1          # print info to files locat.atm & import.cnst
60 TRACK_ATOM         = -1          # ID of particle to track (neg = none)
61 LOAD_BALANCE_FREQ  = 0          # load-balancing intervals (0=no load balancing)
62 SW_LOOKUP_TABLE    = 0          #  use lookup table for SW potential (1=yes,0=no)
```

```
63 LOAD_SIM          = 0        # 0=new sim, 1=continue, 2=use save.n as start
64                              # in order to continue a run,
65                              # the old "simt" and "save.n" must exist!
```

D.2 Compiling the program

The compilation process is fairly straight forward. A *Makefile*[1] is provided, which depending on the target architecture may need to be modified to accommodate the specific compiler. A directory called "Config" contains the architecture specific instructions for the building process. Currently various unix-type platforms are supported (e.g. Linux/i386, SGI Irix/Origin, Sun Solaris/Sparc, etc...). The name of the compiler and the compiler options/flags are given in these target specific configuration files. Before compiling the code, some environment variables must be set to tell 'make' which compiler settings to use. For compiling on a Linux/i366 system, the following commands can be used to set the environment (in BASH[2]):

```
MACHTYPE=i386
VENDOR=intel
OSTYPE=linux
export MACHTYPE VENDOR OSTYPE
```

For convenience, these commands should be added to the `~/.bashrc` file. `make` will then look for a file `./Config/$MACHTYPE-$VENDOR-$OSTYPE.cf`After setting the environment variables and editing the platform specific configuration file (if necessary), the simulation program can be built as follows:

[1]Makefile: A script which tells the Unix program "make" how to build a particular computer program.
[2]BASH (Bourne-Again SHell) is a popular command language interpreter

```
make distclean
make depend
make
```

The first command (`make distclean`) will remove all object files to provide a clean start for the compilation process. `make depend` builds the dependency structure. The last one, `make` compiles the source and builds the simulation program. If changes are made to the code, only the last command needs to be re-executed.

In order to run the program on muliple processors, using MPI, the specify "yes" on line 26 of the `Makefile` (see below). This requires an MPI compiler to be installed on the target system. The file `./Config/Makefile.mpi` may need to be modified to specify the desired compiler. A free MPI implementation (MPICH) for Microsoft Windows or Unix-type systems is available from *http://www-unix.mcs.anl.gov/mpi/mpich/*.

D.2.1 The Makefile

Makefile

```
1  #
2  #       $Id: Makefile,v 1.105 2004/08/26 22:02:59 roman Exp $
3  #
4  ############################################################
5  #
6  #               Where is our project situated?
7  #               If you copy the whole source tree $PWD seems
8  #               to be a good idea. However, sometimes a fix
9  #               path is also good.
10 #
11 TOP        :=        $(PWD)
12 CONFDIR        :=        $(TOP)/Config
13 #
14 ############################################################
15 #
16 #       Answer the following with "yes" or "no".
17 #       Note that the answers are not independent from
18 #       each other, since e.g. some systems do not
19 #       allow otimizing and debugging at the same time!
20 #       BE CAREFUL WITH EXTRA SPACE AT "yes" OR "no"!
```

```
21 #
22 WARN_MODE      := yes
23 OPTIMIZE_MODE  := yes
24 DEBUG_MODE     := no
25 PROFILE_MODE   := no

26 MPI_MODE       := no

27 #
28 ######################################################################
29 #
30 #        Now comes what we want to build.
31 #
32 SRCS  = input.cc laser.cc laser_on.cc silicon.cc  potentl.cc\
33         output.cc calc_par.cc get_par.cc electron.cc\
34         init.cc diamond.cc fcc.cc bcc.cc sc.cc Lattice.cc\
35         load_dat.cc elec_fct.cc\
36         kick.cc bath.cc forces.cc prt_rslt.cc gear.cc track.cc\
37         main.cc utility.cc\
38         State.cc Boxes.cc Colors.cc DataGrid.cc Statistics.cc Transport.cc\
39         HeatFlow.cc Random.cc\
40         ForceTable.cc ForceTableSW.cc StopWatch.cc
41 SRCS_MPI = Communicator.cc Packet.cc PacketForce.cc


42 OBJS  = input.o laser.o laser_on.o silicon.o  potentl.o\
43         output.o calc_par.o get_par.o electron.o\
44         init.o diamond.o fcc.o bcc.o sc.o Lattice.o\
45         load_dat.o elec_fct.o\
46         kick.o bath.o forces.o prt_rslt.o gear.o track.o\
47         main.o utility.o\
48         State.o Boxes.o Colors.o DataGrid.o Statistics.o Transport.o\
49         HeatFlow.o Random.o\
50         ForceTable.o ForceTableSW.o StopWatch.o
51 OBJS_MPI = Communicator.o Packet.o PacketForce.o


52 PROG    = sim

53 #DEPENDINCLUDE = -I/usr/include/g++-v3/

54 #
55 ######################################################
56 #
57 #        Now we are determining our "platform".
58 #        For this we use the environment variables MACHTYPE,
59 #        VENDOR, and OSTYPE. These variables are predefined
60 #        the tcsh. If you do not have the tcsh you should set
61 #        these environment variables by hand.
62 #
63 #        Each "platform" (i.e. = $(MACHTYPE)-$(VENDOR)-$(OSTYPE))
64 #        may offer several compilers. If you do not specify
```

```
65  #        the Variable CS ("compilation system") a default
66  #        value is set.
67  #
68  CS        :=
69  #
70  #
71  include        $(CONFDIR)/$(MACHTYPE)-$(VENDOR)-$(OSTYPE).cf
72  ifeq ($(MPI_MODE),yes)
73          include $(CONFDIR)/Makefile.mpi
74          -include $(CONFDIR)/$(MACHTYPE)-$(VENDOR)-$(OSTYPE)-$(CS)-mpi.cf
75  else
76          include        $(CONFDIR)/$(MACHTYPE)-$(VENDOR)-$(OSTYPE)-$(CS).cf
77          extraFlags := $(CXXFLAGS)
78          CPPFLAGS += $(extraFlags)
79          #CPPFLAGS += $(CXXFLAGS)
80          CXXFLAGS =
81  endif


82  ifeq ($(MPI_MODE),no)
83  CPPFLAGS += -DSIM_WITH_MPI=0
84  endif


85  ifeq ($(MPI_MODE),yes)
86  SRCS += $(SRCS_MPI)
87  OBJS += $(OBJS_MPI)
88  endif



89  #LDOPTIONS += -v

90  #
91  ############################################################
92  #
93  #        Here are the rules. Do not edit!
94  #
95  ############################################################


96  all: $(PROG)


97  $(PROG): $(OBJS)
98          $(RM) $@
99          $(LD) $(LDOPTIONS) -o $@ $(OBJS) $(LDLIBS)
100         @echo ""
101         @echo "$(WARNINGS)"
102         @echo ""
```

```
103 depend:
104         $(DEPEND) $(DEPENDINCLUDE) $(DEPENDSTRING) $(DEPENDFLAGS) -- $(SRCS)

105 doc:
106         doxygen doxygen.config

107 info:
108         @echo "some info"
109         @echo "MACHTYPE = $(MACHTYPE)"
110         @echo "VENDOR = $(VENDOR)"
111         @echo "OSTYPE = $(OSTYPE)"
112         @echo "CS = $(CS)"
113         @echo "CC = $(CC)"
114         @echo "CPP = $(CPP)"
115         @echo "CXX = $(CXX)"
116         @echo "DEPEND = $(DEPEND)"
117         @echo "DEPENDSTRING = $(DEPENDSTRING)"
118         @echo "DEPENDINCLUDE = $(DEPENDINCLUDE)"
119         @echo "DEPENDFLAGS = $(DEPENDFLAGS)"
120         @echo "CPPFLAGS = $(CPPFLAGS)"
121         @echo "CXXFLAGS = $(CXXFLAGS)"
122         @echo "CCFLAGS = $(CCFLAGS)"
123         @echo "CFLAGS = $(CFLAGS)"
124         @echo "LDOPTIONS = $(LDOPTIONS)"
125         @echo "LDLIBS = $(LDLIBS)"
126         @echo "LDFLAGS = $(LDFLAGS)"
127         @echo "that's all folks!"

128 ###########################################################
129 #        Default rules for each Makefile
130 #
131 clean::
132         $(RM)        *.o core
133         $(RM)        -r ii_files

134 distclean: clean
135         $(RM)        $(PROG)

136 again: clean all

137 ###########################################################

138 # DO NOT DELETE THIS LINE -- makedepend depends on it.
```

D.3 Running the program

After (successful) compilation, there will be a file called `sim` in the source directory. Simply execute that executable to run the simulation. The run can be stopped or paused by editing the file `BREAK.prg`. The simulation checks this file on every loop, if the first character is an "n", then the program is continued. To stop the program, set the first character in this file to "y". The simulation can be paused by changing the first character to "s", followed by a number specifying the time to sleep before rechecking the file. These tasks are automated with by the executable scripts `stop`, `pause`, and `resume`.

To run the program in MPI-mode, the executable `sim`, the parameter file, and the file `BREAK.inp` need to be available on each node (or accessable via a networked filesystem). Create a file named `machines` listing the host names of the nodes, one per line (not necessary on the supercomputer[3]). Then the simulation is started with the command

```
mpirun -np <NumProc> -machinefile machines sim
```

where `<NumProc>` is an integer specifying the number of processors (nodes) to be used. More detailed information can be found on the MPICH website or the mpi help file (invoked by the command `man mpi` on Unix-type systems).

D.4 File listing

[3]The simulations were mainly run on SGI Origin machines provided by WestGrid (*http://www.ualberta.ca/CNS/RESEARCH/WestGrid/*)

File	*Description*
Boxes.cc, Boxes.h	data structure for cell subdivision (sec 3.2.3)
Colors.cc, Colors.h	defines some colors for debugging MPI output (different color for different processes)
Communicator.cc, Communicator.h	MPI Interprocess communication
DataGrid.cc, DataGrid.h	data structure for storing 1-D, 2-D, or 3-D data
ForceTable.cc, ForceTable.h	lookup table for particle interaction force and potential
ForceTableSW.cc, ForceTableSW.h	lookup table for Stillinger-Weber force and potential
HeatFlow.cc, HeatFlow.h	1-D heat flow model (with and without electrons or energy buffer)
Lattice.cc, Lattice.h	subroutines to arrange particles into a lattice (e.g. SC, BCC, FCC, diamond)
Packet.cc, Packet.h	data structure for sending particle information between processes (MPI)
PacketForce.cc, PacketForce.h	data structure for sending particle information between processes (MPI). This packs just the forces acting across the node boundaries.

File	*Description*
Random.cc, Random.h	A random number generator (uniform or gaussian distributions)
State.cc, State.h	data structure that holds the particles
Statistics.cc, Statistics.h	data structure and subroutines for averaging temperature and pressure over given periods of (simulation) time
StopWatch.cc, StopWatch.h	a utility to measure time for benchmarking (only wall time for now)
Transport.cc, Transport.h	subroutines to measure transport coefficients
bath.cc, bath.h	subroutines for damping particles (sec. 4.1)
bcc.cc, bcc.h	subroutines for making BCC lattices
calc_par.cc, calc_par.h	calculate further parameters from input parameters
diamond.cc, diamond.h	subroutines for making diamond lattices
elec_fct.cc, elec_fct.h	subroutines for handling electrons: ionization, recombination
electron.cc, electron.h	global variables (constants) related to electrons
fcc.cc, fcc.h	subroutines for making FCC lattices
forces.cc, forces.h	subroutines for interparticle force and potential computation
gear.cc, gear.h	integration routine: Gear's predictor-corrector method (sec. 3.2.2)

File	*Description*
get_par.cc, get_par.h	subroutines for reading parameters from file
init.cc, init.h	initialization routines: position, velocity (temperature)
input.cc, input.h	definition of input variables
kick.cc, kick.h	subroutines for giving energy to atom after decay or neutralization
laser.cc, laser.h	definition of laser related variables
laser_on.cc, laser_on.h	subroutines for simulating the laser pulse (sec. 5.2.1)
load_dat.cc, load_dat.h	subroutine for loading simulation from file (after saving at check-points)
main.cc	contains the main simulation loop and subroutines for setting up the simulation
output.cc, output.h	definition of variables related to saving of simulation data: file names, intervals, etc.
potentl.cc, potentl.h	definition of Stillinger-Weber potential parameters
prt_rslt.cc, prt_rslt.h	subroutines for printing simulation results and check-points (full state of simulation) to files (and terminal).
sc.cc, sc.h	subroutines for making SC lattices
silicon.cc, silicon.h	variable definitions of silicon parameters
track.cc, track.h	subroutines for tracking information on specified particle

File	*Description*
utility.cc, utility.h	miscellaneous utility subroutines
Array2D.h	data structure for 2-D data
Params.h	data structure for storing simulation parameters
Vector3D.h	data structure for storing and manipulating 3-D vector data
arch.h	architecture specific preprocessor commands
const.h	global constants and preprocessor definitions
mympi.h	simply includes the system's mpi++ header (had to hack the system file on the SGI machine to get my code to compile properly, this is no longer necessary)
physics.h	defines physical constants and conversion factors

The following sections present parts of the simulation code. Some of the "uninteresting" portions of the code have been removed (e.g. file I/O, debugging code, etc.).

D.4.1 Main Loop

main.cc

```
1  /** \file main.c
2      \author R.Herrmann R.Holenstein
3      \date   March 1995 - August 1996, 2002-2004

4      \brief Simulation of Ablation of Silicon
5  */


6  #include "const.h"

7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
```

```
10  #include <sys/types.h>
11  #include <unistd.h>
12  #include <time.h>
13  #include <string.h>
14  #include <iostream>
15  #include <iomanip>
16  #include <sstream>
17  #include <cassert>

18  #include "Boxes.h"
19  #include "State.h"
20  #include "Params.h"
21  #include "calc_par.h"
22  #include "electron.h"
23  #include "elec_fct.h"
24  #include "forces.h"
25  #include "gear.h"
26  #include "get_par.h"
27  #include "init.h"
28  #include "input.h"
29  #include "kick.h"
30  #include "bath.h"
31  #include "laser.h"
32  #include "laser_on.h"
33  #include "load_dat.h"
34  #include "output.h"
35  #include "potentl.h"
36  #include "prt_rslt.h"
37  #include "silicon.h"
38  #include "track.h"
39  #include "Statistics.h"
40  #include "HeatFlow.h"
41  #include "Random.h"
42  #include "utility.h"
43  #include "ForceTableSW.h"

44  using namespace std;

45  #define MINI_SIM 0

46  #if SIM_WITH_MPI
47  # include "Communicator.h"
48  # include "Packet.h"
49  #endif

50  const double Math::PI = 4*atan(1.0);

51  #if WITH_COLORS
52  #include "Colors.h"
53  #endif
```

```
54  void printTime( time_t start_time, time_t finish_time );
55  int initialize_atoms(double& time_start, Boxes& pBoxes);
56  int check_break();
57  int startSim();
58  void checkTimeSteps(double aStepSq, double eStepSq);
59  int testRandom();

60  char firstarg[100];
61  void showHelp()
62  {
63    cout << "Usage: " << firstarg
64         << " [-h]"
65         << " [-seed myseed]"
66         << " [-seed2 myseed2]"
67         << " [-testHF]"
68         << " [-testRand]"
69         << " [-testDamp]"
70         << " [-testSWtable]"
71         << " [-genSWtable]"
72         << " [-getLattEng]"
73         << " [-restart]"
74         << " [-thermal]"
75         << " [-dmpTop]"
76         << " [-fixTop]"
77         << " [-fixPressure]"
78         << endl;
79  }


80  # if SIM_WITH_MPI
81  Communicator* comPtr;
82  #endif

83  HeatFlow* hfPtr = 0;


84  bool testLaser = false;
85  bool testHeatFlow = false;
86  bool testRand = false;
87  bool testDamp = false;
88  bool testInt = false;
89  bool testSWtable = false;
90  bool genSWtable = false;
91  bool restartSim = false;
92  bool getHeatCap = false;
93  bool getHeatCond = false;
94  bool getLattEng = false;
95  bool fixTop = false;
96  bool dmpTop = false;
97  bool fixPressure = false;
98  bool mapPotSurf = false;
99  int potSurfMapResX = 20;
```

```
100  int potSurfMapResY = 20;
101  int potSurfMapResZ = 1;

102  void printProgInfo()
103  {
104    cout << "=== HMDSLA 2 ===" << endl;
105    cout << "  compiled: " << __DATE__
106  #       ifdef GCC_VERSION
107          << " (gcc " << GCC_VERSION << ")"
108  #       endif
109          << endl;
110    cout << "  binary representation: " << BINARY_REPRESENTATION << endl;
111    cout << endl << endl;
112  }

113  int main(int argc, char** argv)
114  {
115    int code;
116    time_t start_time, finish_time;

117  # if SIM_WITH_MPI
118    Communicator com(argc, argv);
119    comPtr = &com;
120  # if WITH_COLORS
121    MY_COLOR = Colors::color[com.getId()%(Colors::numColors-2)+2];
122  # endif

123    if (com.getId() == 0)
124        printProgInfo();

125    cout << SET_COLOR << "process " << com.getId()
126         << " on " << com.getProcessorName()
127         << " started" << RESET_COLOR << endl;
128  # else
129    printProgInfo();
130  # endif

131    start_time = time(0);

132    { // write process id to file
133      pid_t procID = getpid();
134      ofstream fout("pid");
135      fout << procID << " " << argv[0] << " started at " << start_time << endl;
136    }

137    int seed1    = 17;
138    int seed2    = 1000;

139    // parse command line arguments
140    strncpy(firstarg, argv[0], 100);
141    for ( int i = 1; i < argc; ++i ) {
142      if ( argv[i] == 0 ) continue;
```

```
143     //cout << "arg[" << i << "] = " << argv[i] << endl;
144     if ( 0 == strcmp( argv[i], "-h" ) ||
145          0 == strcmp( argv[i], "-help" ) ||
146          0 == strcmp( argv[i], "--help" ) ) { showHelp(); return 0; }
147     else if ( 0 == strcmp( argv[i], "-seed"  ) ) sscanf(argv[++i],"%d",&seed1);
148     else if ( 0 == strcmp( argv[i], "-seed2" ) ) sscanf(argv[++i],"%d",&seed2);
149     else if ( 0 == strcmp( argv[i], "-testHF" ) ) testHeatFlow = true;
150     else if ( 0 == strcmp( argv[i], "-testLaser" ) ) testLaser = true;
151     else if ( 0 == strcmp( argv[i], "-testRand" ) ) testRand = true;
152     else if ( 0 == strcmp( argv[i], "-testDamp" ) ) testDamp = true;
153     else if ( 0 == strcmp( argv[i], "-testInt" ) ) testInt = true;
154     else if ( 0 == strcmp( argv[i], "-testSWtable" ) ) testSWtable = true;
155     else if ( 0 == strcmp( argv[i], "-genSWtable" ) ) genSWtable = true;
156     else if ( 0 == strcmp( argv[i], "-getLattEng" ) ) getLattEng = true;
157     else if ( 0 == strcmp( argv[i], "-restart" ) ) restartSim = true;
158     else if ( 0 == strcmp( argv[i], "-heatCap" ) ) getHeatCap = true;
159     else if ( 0 == strcmp( argv[i], "-heatCond" ) ) getHeatCond = true;
160     else if ( 0 == strcmp( argv[i], "-dmpTop" ) ) dmpTop = true;
161     else if ( 0 == strcmp( argv[i], "-fixTop" ) ) fixTop = true;
162     else if ( 0 == strcmp( argv[i], "-fixPressure" ) ) fixPressure = true;
163     else if ( 0 == strcmp( argv[i], "-mapPotSurf" ) ) mapPotSurf = true;
164     else if ( 0 == strcmp( argv[i], "-potSurfRes" ) ) {
165       mapPotSurf = true;
166       sscanf(argv[++i],"%d,%d,%d",
167              &potSurfMapResX,&potSurfMapResY,&potSurfMapResZ);
168     }
169     else {
170       //printf( "no arguments provided, using defaults\n" );
171     }
172   }

173 # if SIM_WITH_MPI

174   bool mpiStatus = true;
175   mpiStatus &= ! MPI_Bcast ( &seed1, 1, MPI_INT, 0, MPI_COMM_WORLD );
176   mpiStatus &= ! MPI_Bcast ( &seed2, 1, MPI_INT, 0, MPI_COMM_WORLD );

177   char arg;

178   arg = (char)testLaser;
179   mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
180   testLaser = (bool)arg;

181   arg = (char)testHeatFlow;
182   mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
183   testHeatFlow = (bool)arg;

184   arg = (char)testRand;
185   mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
186   testRand = (bool)arg;

187   arg = (char)testDamp;
```

```
188     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
189     testDamp = (bool)arg;

190     arg = (char)testInt;
191     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
192     testInt = (bool)arg;

193     arg = (char)testSWtable;
194     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
195     testSWtable = (bool)arg;

196     arg = (char)genSWtable;
197     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
198     genSWtable = (bool)arg;

199     arg = (char)getLattEng;
200     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
201     getLattEng = (bool)arg;

202     arg = (char)getHeatCap;
203     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
204     getHeatCap = (bool)arg;

205     arg = (char)getHeatCond;
206     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
207     getHeatCond = (bool)arg;

208     arg = (char)dmpTop;
209     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
210     dmpTop = (bool)arg;

211     arg = (char)fixTop;
212     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
213     fixTop = (bool)arg;

214     arg = (char)fixPressure;
215     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
216     fixPressure = (bool)arg;

217     arg = (char)mapPotSurf;
218     mpiStatus &= ! MPI_Bcast ( &arg, 1, MPI_CHAR, 0, MPI_COMM_WORLD );
219     mapPotSurf = (bool)arg;

220     if ( !mpiStatus ) {
221       com.exitMessage("failed to distribute command-line arguments");
222       return 1;
223     }
224   # endif


225     /* initialize random function generator */
226     if ( seed1 < 0 ) seed1 = time(0);
```

```
227   if ( seed2 < 0 ) seed2 = time(0);
228 # if SIM_WITH_MPI
229   cout << setw(3) << com.getId() << ": " << flush;
230 # endif
231   cout << "seed1 = " << setw(4) << seed1 << "  --  "
232        << "seed2 = " << setw(4) << seed2 << endl;
233   myRandom.initialize(seed1 , 69069);
234   srand (seed2);


235   /* run simulation */
236   code = startSim();

237   /* print how long the simulation run took to finish */
238   finish_time = time(0);
239   printTime(start_time,finish_time);

240   printf ("--- info: [main.c: main ()] --- end ---\n");

241   return code;
242 }


243 int startSim()
244 {
245 # if SIM_WITH_MPI
246   Communicator& com = *comPtr;
247 # endif

248   /* Initialization */
249   int dum_las , error , far_elec, far_atom;
250   double tf , time_start , atom_time ;
251   char timechk ;
252   bool type_of_iter ;      // atoms = false, electrons = true
253 # if SIM_WITH_MPI
254   bool prev_type_of_iter ; // atoms = false, electrons = true
255 # endif
256   bool next_type_of_iter ; // atoms = false, electrons = true
257   long cntl ;
258   double maxStepSizeElSq = 0;
259   int stopCode = -1;


260 # if SIM_WITH_MPI
261   cout << SET_COLOR
262        << com.getProcessorName() << ":"  << com.getId()
263        << ": MPI communicator initialized"
264        << RESET_COLOR
265        << endl;

266   fileNameAttach = generateFileNameAttach( com.getId() );
267   saveStateFileName += fileNameAttach;
```

```
268     locatAtomFileName += fileNameAttach;

269     string paramsInfoFile = "params-info";
270     paramsInfoFile += fileNameAttach;
271     ofstream infoOut( paramsInfoFile.c_str() );

272   # else
273     ostream& infoOut = cout;
274   # endif

275     first = 1;
276     atoms_gone [0] = 0;
277     atoms_gone [1] = 0;
278     electrons_gone [0] = 0;
279     electrons_gone [1] = 0;

280     dum_las = 0;

281     error = 0;
282     string paramFile = "parameters";
283     if ( ! loadParamFile(paramFile) ) {
284       cout << "could not load file " << paramFile << ", "
285            << "trying old-style PARAM.inp" << endl;
286       error = get_param ();
287     }
288     if (error == 1) {
289       cerr << "could not load parameter file" << endl;
290       return 1;
291     }
292   # if SIM_WITH_MPI
293     if ( com.getId() == 0 )
294   # endif
295     {
296       if ( restartSim ) {
297         //paramFile += ".sav";
298         int cont_prg_ = cont_prg;
299         cont_prg = 1;
300         cout << "saving parameters to file '" << paramFile << "': " << flush;
301         saveParamFile(paramFile);
302         cout << "done." << endl;
303         cont_prg = cont_prg_;
304       }
305     }

306     {
307       string pfile = paramFile;
308   #   if SIM_WITH_MPI
309       pfile += "." + paddedInt(com.getId(),2,'0');
310   #   endif
311       pfile += ".sav";
312       saveParamFile(pfile);
313     }
```

```
314   bool withDamping  = (DIST_DMP>0.0);
315   bool withHeatFlow = (INCL_HF && withDamping);

316 # if SIM_WITH_MPI
317   withHeatFlow = withHeatFlow && com.getId() == 0;
318 # endif

319   fflush (stdout );

320   realtime = 0.;
321   stats_init ();
322   fflush (stdout );

323   if (atom_track >= 0)
324   {
325     string trackAtomFileName = "track.atom" + fileNameAttach;
326     open_track_file(trackAtomFileName);
327   }

328 # if SIM_WITH_MPI
329   { // boxAll only needs to live for a little while
330     Boxes boxAll("boxAll");
331     error = calc_param (params,boxAll);

332     if ( error != 0 ) {
333       com.exitMessage("*** calc_param() returned with error ***",cerr);
334       return 1;
335     }

336     // make sure parallelization is possible:
337     if ( boxAll.numBoxZ <= boxAll.numBoxShare * com.getNumProcessors() ) {
338       if ( com.getId() == 0 )
339         com.exitMessage("*** not enough boxes (in z-dir) per process ***",cerr);
340       return 1;
341     }

342     if ( com.getId() == 0 ) {
343       cout << SET_COLOR << "boxAll:" << RESET_COLOR << endl;
344       boxAll.printInfo();
345     }

346     if ( ! boxes.init(com.getId(), com.getNumProcessors(), params) )
347       return 1;

348     if (cont_prg == 0) { // start from scratch (fresh cluster)

349       if ( com.getId() == 0 )
350         error = initialize_atoms(time_start, boxAll);
351       else
352         error = initialize_atoms(time_start, boxes);
```

```
353       if ( com.getId() == 0 ) {
354         cout << SET_COLOR << "filling boxAll with particles: "
355              << RESET_COLOR << endl;
356         boxAll.fill(state,false); // false=both electrons and atoms
357         bool status = distributeParticles(com, state, boxAll,params);
358         if ( ! com.checkStatus(status) ) {
359           cerr << "*** particle distribution failed ***" << endl;
360           return 1;
361         }
362       }
363       else {
364         bool status = receiveParticles(com, state, boxes, 0);
365         if ( ! com.checkStatus(status) )
366           return 1;
367       }

368       boxes.fill(state,false);
369     }
370     else { // load data from save.n.??
371       error = initialize_atoms(time_start, boxes);
372     }
373   }
374 #else // not SIM_WITH_MPI
375   error = calc_param (params,boxes);
376   if ( error != 0 ) return error;
377   cout << endl << "==== finished calc_param() ====" << endl << endl;
378   cout << "volWidth  = " << params.volWidth << endl;
379   cout << "volHeight = " << params.volHeight << endl;
380   error = initialize_atoms(time_start, boxes);
381   cout << endl << "==== finished initialize_atoms() ====" << endl << endl;
382   if ( error != 0 ) return error;
383 # if MINI_SIM
384   removeAtoms(4);
385 # endif
386   boxes.fill(state,false);
387 #endif // SIM_WITH_MPI


388   // attach marker to output file name (time stamp, process id)
389   removedAtomFileName += fileNameAttach;

390   state.printInfo();
391   boxes.printInfo();

392 # if SIM_WITH_MPI
393   cout << SET_COLOR << "proc " << com.getId() << ":"
394        << boxes.countParticles(boxes.ziMin, boxes.ziMax) << " particles"
395        << " (" << boxes.countParticles() << ")"
396        << RESET_COLOR << endl;

397   com.sync();
398   cout << SET_COLOR << "proc " << com.getId()
```

```
399         << ": now in sync with other processes" << RESET_COLOR << endl;
400     com.sync();
401   # endif

402     print_inputs(infoOut);
403     infoOut << endl << "params info:" << endl; params.print(infoOut);
404     infoOut << endl << "boxes info: " << endl; boxes.printInfo(infoOut);

405     Heatnum = 0;

406     if (error == 1)
407       time_start = TOTTIM + 1.;
408     tim_inc = 1;
409     timechk = 0;

410     fflush (stdout );

411     far_elec = 0;
412     atom_time = time_start - time_st [1] / 2.0;

413     Statistics stats;
414     HeatFlow* heatFlow = 0;
415     if ( hfPtr == 0 ) {
416       if ( withHeatFlow ) heatFlow = new HeatFlow;
417       hfPtr = heatFlow;
418     }
419     else {
420       withHeatFlow = true;
421       heatFlow = hfPtr;
422       heatFlow->printInfo();
423     }
424     double endTimeComputeStats = startSaveStats;
425     double startTimeComputeStats = endTimeComputeStats - statsCollectTime;

426     if ( getLattEng ) {
427   #   if SIM_WITH_MPI
428       cout << "(" << __FILE__ << ":" << __LINE__ << ") "
429            << "this does not work in MPI mode" << endl;
430       return 0;
431   #   endif
432       return getLatticeEnergies(state,boxes,params);
433     }

434     if ( mapPotSurf ) {
435   #   if SIM_WITH_MPI
436       cout << "(" << __FILE__ << ":" << __LINE__ << ") "
437            << "this does not work in MPI mode" << endl;
438       return 0;
439   #   endif
440       return mapPotentialSurface(state,boxes,params,
441                                  potSurfMapResX,potSurfMapResY,potSurfMapResZ);
442     }
```

```
443   ForceTableSW forceTableSW;
444   ForceTableSW forceTableSWbroken;
445   if ( genSWtable ) {
446 #   if SIM_WITH_MPI
447     if ( com.getId() > 0 )
448       return 0;
449 #   endif
450     cout << "generating SW force table" << endl;
451     bool status =
452       generateSWtable(forceTableSW,forceTableSWbroken);
453     cout << "saving tables to files "
454          << "(" << forceTableSW1file << ", " << forceTableSW1file << ")"
455          << endl;
456     status &=
457       forceTableSW.save(forceTableSW1file) &&
458       forceTableSW.save(forceTableSW2file);
459     cout << "done generating force table and saving to file "
460          << (status?"(successful)":"(failed)") << endl;
461     return status;
462   }

463   if ( useForceTable || testSWtable ) {
464     cout << "loading SW force table from files "
465          << "(" << forceTableSW1file << ", " << forceTableSW1file << ")"
466          << endl;
467     if ( !forceTableSW.load(forceTableSW1file) ||
468          !forceTableSWbroken.load(forceTableSW2file) ) {
469       cout << "unable to load tables from files, "
470            << "generating tables instead"
471            << endl;
472       if ( !generateSWtable(forceTableSW,forceTableSWbroken) ) {
473         cerr << "could not load or generate force table" << endl;
474         return 1;
475       }
476     }
477     cout << "force tables enabled" << endl;
478     forceTableSW.printInfo();
479   }



480 # if SIM_WITH_MPI
481   // keep track of load (use wall time as measure)
482   com.stopWatch.reset();
483   com.stopWatch.start();
484 # endif



485   if ( testSWtable ) {
486 #   if SIM_WITH_MPI
```

```
487       if ( com.getId() > 0 )
488         return 0;
489   #   endif
490       return testForceTableSW(forceTableSW, forceTableSWbroken);
491     }

492     if ( testRand ) {
493   #   if SIM_WITH_MPI
494       if ( com.getId() > 0 )
495         return 0;
496   #   endif
497       return testRandom();
498     }

499     if ( testDamp ) {
500   #   if SIM_WITH_MPI
501       if ( com.getId() > 0 )
502         return 0;
503   #   endif
504       return testDamping();
505     }

506     if ( testInt ) {
507   #   if SIM_WITH_MPI
508       if ( com.getId() > 0 )
509         return 0;
510   #   endif
511       return testIntegration(state,boxes);
512     }

513     // test/debug heat-flow model
514     if ( testHeatFlow ) {
515   #   if SIM_WITH_MPI
516       if ( com.getId() > 0 )
517         return 0;
518   #   endif
519       return HeatFlow::test();
520     }

521     if ( testLaser ) {
522       return test_laser();
523     }

524     if ( cont_prg != 1 ) {
525       // only compute forces if not continuing run
526       if ( useForceTable )
527         stopCode += forces( forceTableSW, forceTableSWbroken, 0 );
528       else
529         stopCode += forces ( 0 ); // compute forces incl. SW potential
530     }
```

```
531   stats.computeStats();
532   prt_results(false);

533 # if SIM_WITH_MPI
534   com.sync();
535 # endif


536   //-------------------------------------------------------------------
537   //========================= START OF MAIN LOOP ======================
538   //-------------------------------------------------------------------

539   printf ("=== info: [main.c: main ()] ================");
540   printf ("========= starting main loop ===\n");

541   if (withHeatFlow)
542     hfPtr->setTime(time_start);
543   type_of_iter = false;
544   int itime = 0;
545   for (realtime = time_start ; realtime <= TOTTIM ; realtime += time_st [1] ) {
546 #   if SIM_WITH_MPI
547     prev_type_of_iter = type_of_iter;
548 #   endif
549     // iteration types:
550     //   type_of_iter = true   for electrons
551     //   type_of_iter = false  for atoms
552     type_of_iter = true;
553     next_type_of_iter = true;
554     if (realtime >= atom_time)
555     {
556       type_of_iter = false;
557       atom_time += TIMEST;
558       ++itime;
559     }
560     else if (realtime+time_st [1] >= atom_time) {
561       next_type_of_iter = false;
562     }


563     if (!type_of_iter)
564     {
565       // check if time steps are adequate
566       if (VARIABLE_TIMESTEP)
567         checkTimeSteps(maxStepSizeSq, maxStepSizeElSq);

568       if (timechk == 0)
569         if (realtime >= SPEED_UP1 )
570         {
571           TIMEST  = Speed_inc1 ;
572           calc_param2 (false);
573           timechk = 1;
574         }
```

```
575         if (timechk == 1)
576           if (realtime >= SPEED_UP2 )
577           {
578             TIMEST  = Speed_inc2 ;
579             calc_param2 (false);
580             timechk = 2;
581           }

582         maxStepSizeElSq = 0;
583       }

584       maxStepSizeSq = maxStepSizeElSq;
585       if (Num_el) {
586         far_elec = 0;
587         for (cnt1 = NAT + 1 ; cnt1 <= NAT + elmax ; cnt1 ++)
588         {
589           if (state.ion[cnt1 ] < 0)
590             continue;

591           if ( state.z[cnt1] < boxes.zMinCell || boxes.zMaxCell <= state.z[cnt1] )
592             continue;

593           // predictor step for electrons
594           // ----------------------------
595           far_elec += predict_gear(cnt1, 1);
596         }
597         if (far_elec > 0)
598           cerr << "*** error: [main.c: main ()] " << far_elec
599                << " electron-steps too wide ***" << endl;
600       }

601       maxStepSizeElSq = maxStepSizeSq;

602       if (!type_of_iter )
603       {
604         maxStepSizeSq = 0.0;
605         far_atom = 0;
606         for (cnt1 = Mvnat ; cnt1 <= NAT ; cnt1 ++)
607         {
608   #       if ATOM_TRACK_BEFORE_PREDICT
609           track_particle (cnt1 , "[main.c: main ()]",
610                           "before predictor step (atom)");
611   #       endif

612           if (state.ion [cnt1 ] < 0)
613             continue;

614           if ( state.z[cnt1] < boxes.zMinCell || boxes.zMaxCell <= state.z[cnt1] )
615             continue;

616           // predictor step for atoms
617           // ------------------------
```

```
618           far_atom = predict_gear (cnt1 , 0 );
619         }
620         Numphot = 0;
621         if (far_atom > 0)
622           cerr << "*** error: [main.c: main ()] " << far_elec
623                << " e-steps too wide **** " << endl;
624       }

625   #   if SIM_WITH_MPI
626       if (!type_of_iter ) {
627         cout << SET_COLOR << com.getProcessorName() << ":" << com.getId()
628              << " - count> "
629              << setw(8) << realtime << ":  " << particleCount
630              << RESET_COLOR << endl;

631         if ( realtime == time_start ) {
632           infoOut << endl << endl;
633           infoOut << setw(8) << "time";
634           particleCount.printHeader(infoOut);
635           infoOut << endl;
636         }
637         infoOut << setw(8) << realtime;
638         particleCount.print(infoOut);
639         infoOut << endl;

640         particleCount.clear();
641       }
642   #   endif

643   #   if SIM_WITH_MPI
644       com.stopWatch.stop();
645       communicate(com,state,boxes, type_of_iter && prev_type_of_iter);

646       if (doLoadBalancing && !type_of_iter && itime % loadBalanceFreq == 0 ) {
647         loadBalancing(com,state,boxes);
648       }

649       save_state_backup();
650       com.stopWatch.start();
651   #   endif

652       // if neither current nor next iteration is for atoms,
653       // then fill boxes with electrons only
654       // otherwise fill with all particles (atoms and electrons)
655       boxes.fill(state, type_of_iter && next_type_of_iter);

656       // check for laser
657       // ---------------
658       if (!type_of_iter )
659       {
660         if ((dum_las == 0) && (realtime > LAS_ON )) {
661           dum_las = 1;
```

```
662         }
663         else if ((dum_las == 1) && (realtime > Las_Off )) {
664           dum_las = 2;
665         }

666         if ( dum_las == 1 )
667           laser_on (realtime );

668         for (cnt1 = Mvnat ; cnt1 <= NAT ; cnt1 ++)
669         {
670           if (state.ion[cnt1] < 0 || state.exc[cnt1] < 0) continue;

671           if ( state.z[cnt1] < boxes.zMinCell || boxes.zMaxCell <= state.z[cnt1] )
672             continue;

673           if (state.exc[cnt1] >= 1)
674           {
675             stopCode += decay_atom (cnt1);
676           }
677           if (state.ion[cnt1] > 0)
678           {
679             tf = exp (- TIMEST / IONRECOM);
680             if (myRandom.rand() > tf)
681             {
682               state.ion[cnt1] --;
683               state.exc[cnt1] = Ionnumphot - 1;
684             }
685           }
686         }
687       }

688       EKIN_prev = state.kinE;
689       EPOT_prev = state.potE;

690       if ( useForceTable )
691         forces( forceTableSW, forceTableSWbroken, type_of_iter );
692       else
693         forces ( type_of_iter ); // compute forces incl. SW potential

694   #   if SIM_WITH_MPI
695       if ( incl_StiWeb && !type_of_iter) {
696         com.stopWatch.stop();
697         communicate_forces(com,state,boxes);
698         com.stopWatch.start();
699       }
700   #   endif

701       /* reset counters */
702       Num_el = 0;

703       for (cnt1 = NAT + 1; cnt1 <= NAT + elmax; cnt1 ++)
704       {
```

```
705       if (state.ion [cnt1 ] < 0)
706         continue;

707       if ( state.z[cnt1] < boxes.zMinCell || boxes.zMaxCell <= state.z[cnt1] )
708         continue;

709       if (state.ion [cnt1 ] > 0)
710         Num_el ++;

711       if (!Num_el)
712         continue;

713       // corrector step for electrons
714       // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
715       correct_gear (cnt1 , 1);
716     }

717     if (!type_of_iter) {
718       if ( equilibrate && realtime < lastEquilibTime ) {
719         cout << "=== equilibration: damping particles to "
720              << TEMP << "K ===" << endl;
721         for (int pid = Mvnat ; pid <= NAT ; pid ++) {
722           if ( state.ion[pid] < 0 ) continue;
723           dampParticle_force(state, pid, TEMP);
724         }
725       }

726       Numexc = 0;
727       Numion = 0;
728       Numabl = 0;
729       int pCnt = 0;
730       for (cnt1 = Mvnat ; cnt1 <= NAT ; cnt1 ++)
731       {
732         if (state.ion [cnt1 ] < 0)
733         {
734           Numabl ++;
735           continue;
736         }

737         if ( state.z[cnt1] < boxes.zMinCell || boxes.zMaxCell <= state.z[cnt1] )
738           continue;

739         if (state.ion [cnt1 ] > 0)
740           Numion ++;

741         if (state.exc [cnt1 ] > 0)
742           Numexc ++;


743         if (withDamping && cnt1 < Mvnat_nd) /* Langevin damping */
744         {
745           if (withHeatFlow)
```

```
746             heatFlow->dampParticle(state, cnt1);
747           else
748             dampParticle(state, cnt1, TEMP);
749         }
750         //  corrector step for atoms
751         //  ------------------------
752         correct_gear (cnt1 , 0);

753         // --- kinetic energy ---
754         state.kinE += state.vx[cnt1 ] * state.vx[cnt1 ];
755         state.kinE += state.vy[cnt1 ] * state.vy[cnt1 ];
756         state.kinE += state.vz[cnt1 ] * state.vz[cnt1 ];
757         ++pCnt;

758 #       if ATOM_TRACK_AFTER_CORRECT
759         if (atom_track >= 0)
760           track_particle (cnt1 , "[main.c: main ()]",
761                           "after corrector step (atom)");
762 #       endif


763       }
764       if (atom_track >= 0)
765         fflush (ftrack );


766       // kinetic energy transformed from Ang^2 into eV
767       state.kinE *= 0.5 * state.massAt;
768       RMvnat = pCnt;

769       if ( withHeatFlow ) {
770         if ( boxes.firstNonDamped >= 0 ) {
771           if ( equilibrate && realtime < lastEquilibTime ) {
772             heatFlow->setTemperatureTop(TEMP, time_st[0]);
773           }
774           else {
775             int numLayers = HF_COUPLE_NUMLAYERS_TEMP_AVE;
776             int ziStart = boxes.firstNonDamped;
777             int ziEnd = ziStart + numLayers - 1;
778             double temp = state.getTemperature(boxes, ziStart, ziEnd);
779             heatFlow->setTemperatureTop(temp, time_st[0]);
780 #           endif
781           }
782           if ( ! heatFlow->step() ) {
783             stopCode = 1;
784 #           if SIM_WITH_MPI
785             com.exitMessage("Heat flow model aborted",true);
786 #           endif
787           }
788         }
789       }
```

```
790       if ( realtime >= startTimeComputeStats ) {
791         if ( realtime >= endTimeComputeStats ) {
792           stats.computeStats(true); // finalize stats
793           stats.save();
794           print_heatflow(heatFlow);
795           endTimeComputeStats = realtime + saveStatsInterval - time_st [1];
796           startTimeComputeStats = endTimeComputeStats - statsCollectTime;
797         }
798         else
799           stats.computeStats();
800       }
801     }


802 #   if SIM_WITH_MPI
803     if (!type_of_iter &&
804         ((realtime >= cnt_save ) || (realtime >= TOTTIM ) || (first == 1)) )
805       particleInventory(com, particleCount, boxes);
806 #   endif

807     prt_results (type_of_iter );
808     fflush (stdout );

809     if ( stopCode == -1 )
810       stopCode = check_break();

811 #   if SIM_WITH_MPI
812     stopCode = ( com.checkStatus(stopCode == -1) ? -1 : 1 );
813 #   endif

814     if ( stopCode != -1 ) {
815 #     if SIM_WITH_MPI
816       if ( com.getId() == 0 )
817 #     endif
818         cout << endl << endl << "(" << __FILE__ << ":" << __LINE__ << ") "
819              << "*** aborting program (" << stopCode << ") ***"
820              << endl << endl << endl;
821       return stopCode;
822     }

823 #   if ! SIM_WITH_MPI
824     // can't take this shortcut anymore in MPI_MODE
825     if ((Num_el <= 0) && (type_of_iter == false))
826       realtime += time_st [0] - time_st [1];
827 #   endif
828   }

829   final_output ();

830   return 0;         /* main should return something        */
831 }
```

D.4.2 Integrator

gear.cc

```
/**
   \brief do predictor step for a particle
   \param pid: number of particle
   \param delta: 0 = atoms, 1 = electrons
   \return error code (1 = electron step too far)
*/
int predict_gear (long pid , int delta )
{
  // Gear's 5th order predictor

  int error = 0;
  const double timeStep = time_st[delta];

  // predict location of particle

  state.vx[pid] *= timeStep;
  state.vy[pid] *= timeStep;
  state.vz[pid] *= timeStep;

  double tdblx  = state.vx[pid] + state.ax[pid] + state.bx[pid] +
                  state.cx[pid] + state.dx[pid];
  double tdbly  = state.vy[pid] + state.ay[pid] + state.by[pid] +
                  state.cy[pid] + state.dy[pid];
  double tdblz  = state.vz[pid] + state.az[pid] + state.bz[pid] +
                  state.cz[pid] + state.dz[pid];

  state.x[pid] += tdblx;
  state.y[pid] += tdbly;
  state.z[pid] += tdblz;

  state.vx[pid] += 2*state.ax[pid] + 3*state.bx[pid] +
                   4*state.cx[pid] + 5*state.dx[pid];
  state.vy[pid] += 2*state.ay[pid] + 3*state.by[pid] +
                   4*state.cy[pid] + 5*state.dy[pid];
  state.vz[pid] += 2*state.az[pid] + 3*state.bz[pid] +
                   4*state.cz[pid] + 5*state.dz[pid];

  state.ax[pid] += 3*state.bx[pid] + 6*state.cx[pid] + 10*state.dx[pid];
  state.ay[pid] += 3*state.by[pid] + 6*state.cy[pid] + 10*state.dy[pid];
  state.az[pid] += 3*state.bz[pid] + 6*state.cz[pid] + 10*state.dz[pid];

  state.bx[pid] +=                  4*state.cx[pid] + 10*state.dx[pid];
  state.by[pid] +=                  4*state.cy[pid] + 10*state.dy[pid];
  state.bz[pid] +=                  4*state.cz[pid] + 10*state.dz[pid];

  state.cx[pid] +=                                    5*state.dx[pid];
  state.cy[pid] +=                                    5*state.dy[pid];
  state.cz[pid] +=                                    5*state.dz[pid];
```

```
40    state.vx[pid] /= timeStep;
41    state.vy[pid] /= timeStep;
42    state.vz[pid] /= timeStep;


43    // keep track of the maximum step size
44    if (VARIABLE_TIMESTEP) {
45      double distSq = tdblx*tdblx + tdbly*tdbly + tdblz*tdblz;
46      if ( distSq > maxStepSizeSq )
47        maxStepSizeSq = distSq;
48    }

49    // check change in location

50    if (pid < state.firstElIndex)
51    {
52      if ((tdblx > XSTP ) || (tdblx < -XSTP ) ||
53          (tdbly > XSTP ) || (tdbly < -XSTP ) ||
54          (tdblz > XSTP ) || (tdblz < -XSTP )) {
55  #     if VERBOSE_PREDICT_GEAR
56        cerr << realtime << ": atom step size too large (" << pid << "): "
57             << sqrt(tdblx*tdblx + tdbly*tdbly + tdblz*tdblz)
58             << endl;
59  #     endif
60        ++error;
61      }
62    }
63    else {
64      if ((tdblx > EXSTP ) || (tdblx < -EXSTP ) ||
65          (tdbly > EXSTP ) || (tdbly < -EXSTP ) ||
66          (tdblz > EXSTP ) || (tdblz < -EXSTP )) {
67  #     if VERBOSE_PREDICT_GEAR
68        cerr << realtime << ": electron step size too large (" << pid << "): "
69             << sqrt(tdblx*tdblx + tdbly*tdbly + tdblz*tdblz)
70             << endl;
71  #     endif
72        ++error;
73      }
74    }

75    // check if particle is still close to bulk
76    checkPosition(pid);

77    return (error);
78  }


79  /**
80   *  \brief do corrector step for a particle
81   *  \param pid    number of particle
82   *  \param delta  0 use parameters [0] (atoms),
83   *                1 use parameters [1] (electrons)
```

```
84   */
85  void correct_gear (long pid , int delta )
86  {
87    // Gear's 5th order corrector
88    static const double  A0 =   3.0/20.0;
89    static const double  A1 = 251.0/360.0;
90    static const double  A3 =  11.0/18.0;
91    static const double  A4 =   1.0/6.0;
92    static const double  A5 =   1.0/60.0;

93    const double timeStep = time_st[delta];
94    const double step = 0.5 * timeStep * timeStep;

95    state.vx[pid] *= timeStep;
96    state.vy[pid] *= timeStep;
97    state.vz[pid] *= timeStep;

98    double ax = step * state.fx[pid] * imass[delta];
99    double ay = step * state.fy[pid] * imass[delta];
100   double az = step * state.fz[pid] * imass[delta];

101   double dax = ax - state.ax[pid];
102   double day = ay - state.ay[pid];
103   double daz = az - state.az[pid];

104   state.x[pid] += A0*dax;
105   state.y[pid] += A0*day;
106   state.z[pid] += A0*daz;

107   state.vx[pid] += A1*dax;
108   state.vy[pid] += A1*day;
109   state.vz[pid] += A1*daz;

110   state.ax[pid]  = ax;
111   state.ay[pid]  = ay;
112   state.az[pid]  = az;

113   state.bx[pid] += A3*dax;
114   state.by[pid] += A3*day;
115   state.bz[pid] += A3*daz;

116   state.cx[pid] += A4*dax;
117   state.cy[pid] += A4*day;
118   state.cz[pid] += A4*daz;

119   state.dx[pid] += A5*dax;
120   state.dy[pid] += A5*day;
121   state.dz[pid] += A5*daz;


122   state.vx[pid] /= timeStep;
123   state.vy[pid] /= timeStep;
```

```
124   state.vz[pid] /= timeStep;

125   checkPosition(pid);
126 }
```

D.4.3 Force Calculation

forces.cc

```
1  /**
2     \brief calculate TWO BODY POTENTIAL OF SYSTEM
3     \param xij   x-component of vector between atom 1 and 2
4     \param yij   y-component of vector between atom 1 and 2
5     \param zij   z-component of vector between atom 1 and 2
6     \param cnt1  number of first atom
7     \param cnt2  number of second atom
8  */
9  static
10 void  two_pot (double  xij, double  yij, double  zij, long cnt1 , long cnt2 )
11 {
12   //return; assert(false);

13   double  potv , sq_dist , dist , fc , fcd , r , potvd , tflt ;

14   potv = 0.;

15   sq_dist = xij * xij + yij * yij + zij * zij;
16   dist    = sqrt (sq_dist);
17   r       = dist - R_CUTOFF ;

18   fc   = exp (D_CUTOFF / r);
19   fcd  = - D_CUTOFF * fc / (r * r * dist);
20   potv = A1 / (sq_dist * sq_dist);

21   // is bond broken ??
22   tflt = (double) state.exc [cnt1 ];
23   if (!incl_brkbnd || myRandom.rand() >= tflt / 4. )
24     potv -= A2;
25   //else cout << "broken bond: " << cnt1 << endl;

26   potvd = - 4.0 * A1 / (sq_dist * sq_dist * sq_dist);

27   // 0.5 from i<j & i>j
28   double f = -0.5 * (fcd * potv + fc * potvd);

29   double fx = f * xij;
```

```
30    double fy = f * yij;
31    double fz = f * zij;

32    state.fx[cnt1] += fx;
33    state.fy[cnt1] += fy;
34    state.fz[cnt1] += fz;

35    state.fx[cnt2] -= fx;
36    state.fy[cnt2] -= fy;
37    state.fz[cnt2] -= fz;

38    double vir = f * sq_dist;

39    if ((cnt1 == atom_track) || (cnt2 == atom_track))
40      printf ("state.fx [%ld] = %f, state.fx [%ld] = %f \n", cnt1 , state.fx [cnt1 ],
41              cnt2 , state.fx [cnt2 ]);

42    potv  *= fc * 0.5;
43    state.potE  += potv;
44    state.virS  += vir;


45    potv  *= 0.5;
46    state.pot[cnt1] += potv;
47    state.pot[cnt2] += potv;

48    vir  *= 0.5;
49    state.virial[cnt1] += vir;
50    state.virial[cnt2] += vir;

51  # if MEASURE_CONDUCTIVITY
52    double fv1 = fx * state.vx[cnt1] + fy * state.vy[cnt1] + fz * state.vz[cnt1];
53    state.rfv[cnt1].x += (-xij)*fv1;
54    state.rfv[cnt1].y += (-yij)*fv1;
55    state.rfv[cnt1].z += (-zij)*fv1;

56    double fv2 = (-fx) * state.vx[cnt2] + (-fy) * state.vy[cnt2] + (-fz) * state.vz[cnt2];
57    state.rfv[cnt2].x += xij*fv2;
58    state.rfv[cnt2].y += yij*fv2;
59    state.rfv[cnt2].z += zij*fv2;
60  # endif

61    return ;
62  }


63  /**
64     \brief calculate THREE BODY POTENTIAL OF SYSTEM
65     \param xij   x-component of vector between atom 1 and 2
66     \param yij   y-component of vector between atom 1 and 2
67     \param zij   z-component of vector between atom 1 and 2
68     \param i     number of first atom
```

```
69      \param j      number of second atom
70  */
71  static
72  void three_pot (const double xij, const double yij, const double zij,
73                  const long i, const long j)
74  {
75    //return; assert(false);
76    double sq_dist12 , dist12 , sq_dist13 , dist13;
77    double rjkc2;
78    double potv, virS, tpot, tvir;
79    double xik, yik, zik, rqijk;
80    double wd_xj, wd_yj, wd_zj, wd_xk, wd_yk, wd_zk;
81    double u, ud, v, vd, w, wq, rj_R, rk_R;
82    double udvwq, uvdwq, uv;
83    double fxj, fyj, fzj;
84    double fxk, fyk, fzk;

85    int cntx3, cnty3, cntz3, cntc3;
86    int cntx3i, cnty3i, cntz3i;

87    long cnt3;

88  # if MEASURE_CONDUCTIVITY
89    double fv;
90  # endif

91    potv = 0.;
92    virS = 0.;

93    sq_dist12 = xij * xij + yij * yij + zij * zij;
94    dist12    = sqrt (sq_dist12 );

95    rj_R   = dist12 - R_CUTOFF ;
96    u      = exp (ALPHA * D_CUTOFF / rj_R);
97    ud     = u * D_CUTOFF * ALPHA / (rj_R * rj_R * dist12);

98    /* 3rd loop  */
99    for (cntx3i = x3start; cntx3i <= x3stop; cntx3i++) {
100     cntx3 = cntx3i;
101 #   if PERIODIC_BOUNDARIES_X
102     if      (cntx3 <       0) cntx3 += boxes.numBoxXY;
103     else if (cntx3 >= boxes.numBoxXY) cntx3 -= boxes.numBoxXY;
104 #   endif
105     for (cnty3i = y3start; cnty3i <= y3stop; cnty3i++) {
106       cnty3 = cnty3i;
107 #     if PERIODIC_BOUNDARIES_Y
108       if      (cnty3 <       0) cnty3 += boxes.numBoxXY;
109       else if (cnty3 >= boxes.numBoxXY) cnty3 -= boxes.numBoxXY;
110 #     endif
111       for (cntz3i = z3start; cntz3i <= z3stop; cntz3i++) {
112         cntz3 = cntz3i;
113 #       if PERIODIC_BOUNDARIES_Z
```

```
114         if      (cntz3 <       0) cntz3 += boxes.numBoxZ;
115         else if (cntz3 >= boxes.numBoxZ) cntz3 -= boxes.numBoxZ;
116 #       endif
117         for (cntc3 = 0; cntc3 < boxes.count[cntx3] [cnty3] [cntz3]; cntc3++)
118         {
119           cnt3 = boxes.box[cntx3] [cnty3] [cntz3] [cntc3];

120           if (cnt3 <= j)
121             continue;
122           if (state.ion [cnt3 ] < 0)
123             continue;
124           if (cnt3 > NAT )
125             continue;
126           if ((state.ion [cnt3 ]) && (state.ion [j ]))
127             continue;
128           if ((state.ion [cnt3 ]) && (state.ion [i ]))
129             continue;

130           if ((cnt3 != i) && (cnt3 != j))
131           {
132             xik = state.x[i ] - state.x[cnt3 ];
133             yik = state.y[i ] - state.y[cnt3 ];
134             zik = state.z[i ] - state.z[cnt3 ];
135 #           if PERIODIC_BOUNDARIES_X
136             while ( xik >  params.volHalfWidth ) xik -= params.volWidth;
137             while ( xik < -params.volHalfWidth ) xik += params.volWidth;
138 #           endif
139 #           if PERIODIC_BOUNDARIES_Y
140             while ( yik >  params.volHalfWidth ) yik -= params.volWidth;
141             while ( yik < -params.volHalfWidth ) yik += params.volWidth;
142 #           endif
143 #           if PERIODIC_BOUNDARIES_Z
144             while ( zik >  params.volHalfHeight ) zik -= params.volHeight;
145             while ( zik < -params.volHalfHeight ) zik += params.volHeight;
146 #           endif

147             sq_dist13   =   xik * xik + yik * yik + zik * zik;
148             if  (sq_dist13   < RCSQ)
149             {
150               dist13 = sqrt (sq_dist13  );

151               /* calculate cos theta */
152               rqijk  = xij * xik + yij * yik + zij * zik;
153               rjkc2  = 2.0 / (dist12 * dist13);                  // *2.0 s.b.

154               w       = rqijk / dist13 / dist12 - costetan;
155               wq      = w * w; // wq = w quadrat = w squared

156               rjkc2  *= w;                                       //  *w s.b.

157               wd_xj   = rjkc2 * (rqijk / sq_dist12 * xij - xik );
158               wd_yj   = rjkc2 * (rqijk / sq_dist12 * yij - yik);
```

```
159             wd_zj   = rjkc2 * (rqijk / sq_dist12 * zij - zik);

160             wd_xk   = rjkc2 * (rqijk / sq_dist13 * xik - xij );
161             wd_yk   = rjkc2 * (rqijk / sq_dist13 * yik - yij);
162             wd_zk   = rjkc2 * (rqijk / sq_dist13 * zik - zij);

163             rk_R    = dist13 - R_CUTOFF ;
164             v       = exp (ALPHA * D_CUTOFF / rk_R);
165             vd      = v * D_CUTOFF * ALPHA / (rk_R * rk_R * dist13);

166             uv     = ZETT * u * v;
167             udvwq = ZETT * ud * v * wq;
168             uvdwq = ZETT * u * vd * wq;

169             tpot  = uv * wq;

170             fxj = (uv * wd_xj + udvwq * xij);
171             fyj = (uv * wd_yj + udvwq * yij);
172             fzj = (uv * wd_zj + udvwq * zij);

173             fxk = (uv * wd_xk + uvdwq * xik);
174             fyk = (uv * wd_yk + uvdwq * yik);
175             fzk = (uv * wd_zk + uvdwq * zik);

176             state.fx [i] += (fxj + fxk);
177             state.fy [i] += (fyj + fyk);
178             state.fz [i] += (fzj + fzk);

179             state.fx [j] -= fxj;
180             state.fy [j] -= fyj;
181             state.fz [j] -= fzj;

182             state.fx [cnt3] -= fxk;
183             state.fy [cnt3] -= fyk;
184             state.fz [cnt3] -= fzk;

185             state.pot [i] += tpot;
186             potv += tpot;

187             // double negative because xij is actually xji
188             // (i.e. sum adds negative)
189             tvir = (fxj*xij + fyj*yij + fzj*zij);
190             virS += tvir;
191             tvir *= 0.5;
192             state.virial[i] += tvir;
193             state.virial[j] += tvir;

194             tvir = -(fxk*xik + fyk*yik + fzk*zik);
195             virS += tvir;
196             tvir *= 0.5;
197             state.virial[i] += tvir;
198             state.virial[cnt3] += tvir;
```

```
199 #             if MEASURE_CONDUCTIVITY
200               fv = (fxj) * state.vx[i]
201                 +  (fyj) * state.vy[i]
202                 +  (fzj) * state.vz[i];
203               state.rfv[i].x += (-xij)*fv;
204               state.rfv[i].y += (-yij)*fv;
205               state.rfv[i].z += (-zij)*fv;

206               fv = (fxk) * state.vx[i]
207                 +  (fyk) * state.vy[i]
208                 +  (fzk) * state.vz[i];
209               state.rfv[i].x += (-xik)*fv;
210               state.rfv[i].y += (-yik)*fv;
211               state.rfv[i].z += (-zik)*fv;


212               fv = (-fxj) * state.vx[j]
213                 +  (-fyj) * state.vy[j]
214                 +  (-fzj) * state.vz[j];
215               state.rfv[j].x += (xij)*fv;
216               state.rfv[j].y += (yij)*fv;
217               state.rfv[j].z += (zij)*fv;

218               fv = (-fxk) * state.vx[cnt3]
219                 +  (-fyk) * state.vy[cnt3]
220                 +  (-fzk) * state.vz[cnt3];
221               state.rfv[cnt3].x += (xik)*fv;
222               state.rfv[cnt3].y += (yik)*fv;
223               state.rfv[cnt3].z += (zik)*fv;
224 #             endif

225 #             if 0
226               cout << " dist12=" << dist12
227                    << " dist13=" << dist13
228                    << " rj_R=" << rj_R
229                    << " rk_R=" << rk_R
230                    << " ALPHA=" << ALPHA
231                    << " D_CUTOFF=" << D_CUTOFF
232                    << " R_CUTOFF=" << R_CUTOFF
233                    << " ZETT=" << ZETT
234                    << endl;
235 #             endif

236               if ((i == atom_track) || (j == atom_track) || (cnt3 == atom_track))
237                 printf ("fx[%ld] = %f, fx[%ld] = %f, fx[%ld] = %f \n",
238                         i , state.fx [i ], j , state.fx [j ], cnt3 , state.fx [cnt3 ]);

239               //cout << "+" << flush;
240             }
241           }
242         }
```

```
243         }
244       }
245     }

246     state.potE  += potv;
247     state.virS  += virS;
248   }



249   int coulomb(int cnt1, int boxX, int boxY, int boxZ, vector<int>& coulombNeighbours)
250   {
251     double rdsq;
252     double xij, yij, zij;
253     double fx, fy, fz;
254     double fcoul, rdsr;
255     double pot,vir;
256     int cntx2, cnty2, cntz2, cntc2 ;
257     int cntx2i, cnty2i, cntz2i;
258     long cnt2;
259     long ltint1 = 0;
260     long ltint2 = 0;
261     int error = 0;
262   # if MEASURE_CONDUCTIVITY
263     double fv;
264   # endif

265     x2start = boxX - boxes.numBoxShare;
266     x2stop  = boxX + boxes.numBoxShare;
267     y2start = boxY - boxes.numBoxShare;
268     y2stop  = boxY + boxes.numBoxShare;
269     z2start = boxZ - boxes.numBoxShare;
270     z2stop  = boxZ + boxes.numBoxShare;

271   # if ! PERIODIC_BOUNDARIES_X
272     if (x2start <       0) x2start = 0;
273     if (x2stop  >= boxes.numBoxXY) x2stop  = boxes.numBoxXY - 1;
274   # endif
275   # if ! PERIODIC_BOUNDARIES_Y
276     if (y2start <       0) y2start = 0;
277     if (y2stop  >= boxes.numBoxXY) y2stop  = boxes.numBoxXY - 1;
278   # endif
279   # if ! PERIODIC_BOUNDARIES_Z
280     if (z2start <       0) z2start = 0;
281     if (z2stop  >= boxes.numBoxZ) z2stop  = boxes.numBoxZ - 1;
282   # endif


283     // 2nd loop Coulomb
284     for (cntx2i = x2start; cntx2i <= x2stop; cntx2i++) {
285       cntx2 = cntx2i;
286   #   if PERIODIC_BOUNDARIES_X
```

```
287     if      (cntx2 <       0) cntx2 += boxes.numBoxXY;
288     else if (cntx2 >= boxes.numBoxXY) cntx2 -= boxes.numBoxXY;
289 #   endif
290     for (cnty2i = y2start; cnty2i <= y2stop; cnty2i++) {
291       cnty2 = cnty2i;
292 #     if PERIODIC_BOUNDARIES_Y
293       if      (cnty2 <       0) cnty2 += boxes.numBoxXY;
294       else if (cnty2 >= boxes.numBoxXY) cnty2 -= boxes.numBoxXY;
295 #     endif
296       for (cntz2i = z2start; cntz2i <= z2stop; cntz2i++) {
297         cntz2 = cntz2i;
298 #       if PERIODIC_BOUNDARIES_Z
299         if      (cntz2 <       0) cntz2 += boxes.numBoxZ;
300         else if (cntz2 >= boxes.numBoxZ) cntz2 -= boxes.numBoxZ;
301 #       endif

302         for (cntc2 = 0; cntc2 < boxes.count[cntx2] [cnty2] [cntz2]; cntc2++) {
303 /*3*/
304           cnt2 = boxes.box[cntx2] [cnty2] [cntz2] [cntc2];

305           if (state.ion [cnt2 ] < 0)
306             continue;

307           if (cnt1 == cnt2)
308             continue;


309           // Coulomb potential
310           // ~~~~~~~~~~~~~~~~~~
311           if (state.ion [cnt1 ] > 0)
312             if (state.ion [cnt2 ] > 0) {
313 /*4*/
314               xij =   state.x[cnt1] - state.x[cnt2];
315               yij =   state.y[cnt1] - state.y[cnt2];
316               zij =   state.z[cnt1] - state.z[cnt2];

317 #             if PERIODIC_BOUNDARIES_X
318               if      ( xij >  params.volHalfWidth ) xij -= params.volWidth;
319               else if ( xij < -params.volHalfWidth ) xij += params.volWidth;
320 #             endif
321 #             if PERIODIC_BOUNDARIES_Y
322               if      ( yij >  params.volHalfWidth ) yij -= params.volWidth;
323               else if ( yij < -params.volHalfWidth ) yij += params.volWidth;
324 #             endif
325 #             if PERIODIC_BOUNDARIES_Z
326               if      ( zij >  params.volHalfHeight ) zij -= params.volHeight;
327               else if ( zij < -params.volHalfHeight ) zij += params.volHeight;
328 #             endif

329               rdsq = xij * xij + yij * yij + zij * zij;

330               if (rdsq >= CLRCSQ)
```

```
331                 continue;

332               fcoul = 1.0;
333               if (cnt1 > NAT )
334               {
335                 fcoul *= -1.;
336                 ltint2 = cnt1 ;
337                 ltint1 = cnt2 ;
338               }
339               if (cnt2 > NAT )
340               {
341                 fcoul *= -1.;
342                 ltint1 = cnt1 ;
343                 ltint2 = cnt2 ;
344               }

345               rdsr = sqrt (rdsq);
346               pot = fcoul * 0.14399762e2 * state.ion [cnt1 ] * state.ion [cnt2 ] / rdsr;
347               fcoul *= pot / rdsq;

348               if (fcoul < 0.0 && rdsq < recombine_radiusq ) // minimum radius for e-i+
349                 error += recombine ( ltint1 , ltint2 );
350               else
351                 state.pot [cnt1] += pot;

352               // 0.5 for i<j and i>j (not efficient, i know, just keeping it the way it was)
353               fx = 0.5 * (xij * fcoul);
354               fy = 0.5 * (yij * fcoul);
355               fz = 0.5 * (zij * fcoul);

356               state.fx [cnt1] += fx;
357               state.fy [cnt1] += fy;
358               state.fz [cnt1] += fz;

359               state.fx [cnt2] -= fx;
360               state.fy [cnt2] -= fy;
361               state.fz [cnt2] -= fz;

362               vir = -fcoul * rdsq;
363               state.virS  += vir;
364               vir  *= 0.5;
365               state.virial[cnt1] += vir;
366               state.virial[cnt2] += vir;


367 #             if MEASURE_CONDUCTIVITY
368               fv = fx * state.vx[cnt1]
369                 +  fy * state.vy[cnt1]
370                 +  fz * state.vz[cnt1];
371               state.rfv[cnt1].x += (-xij)*fv;
372               state.rfv[cnt1].y += (-yij)*fv;
373               state.rfv[cnt1].z += (-zij)*fv;
```

```
374                 fv = (-fx) * state.vx[cnt2]
375                   +  (-fy) * state.vy[cnt2]
376                   +  (-fz) * state.vz[cnt2];
377                 state.rfv[cnt2].x += xij*fv;
378                 state.rfv[cnt2].y += yij*fv;
379                 state.rfv[cnt2].z += zij*fv;
380 #               endif

381 /*4*/
382               }

383             coulombNeighbours.push_back(cnt2);
384 /*3*/
385           }
386         }
387       }
388     }
389     return error;
390 }


391 /**
392    \brief calculate Forces according to TERSOFF and Coulomb potential
393    \param sti_web  if = 1 no Stillinger Weber potential added
394 */
395 int forces (int sti_web )
396 {

397 # if BENCHMARK_FORCES
398   StopWatch clock;
399   clock.reset();
400   clock.start();
401 # endif

402   if ( incl_StiWeb == 0 )
403     sti_web = 1;

404   //test_twoPot();

405   double  xij, yij, zij;
406   double  fcoul, rdsr, vir;

407   int cntc, cntx, cnty, cntz ;
408   int cntx2, cnty2, cntz2, cntc2 ;
409   int cntx2i, cnty2i, cntz2i;
410   double rdsq , tflt, pot ;
411   long cnt1 , cnt2;
412   long ltint1 = 0;
413   long ltint2 = 0;
414   int error = 0;
415 # if MEASURE_CONDUCTIVITY
```

```
416     double fv;
417   # endif

418     for (cnt1 = Mvnat; cnt1 <= NAT + elmax; cnt1++) {
419       state.fx [cnt1] = 0.;
420       state.fy [cnt1] = 0.;
421       state.fz [cnt1] = 0.;
422       state.pot [cnt1] = 0.;
423       state.virial [cnt1] = 0.;
424   #   if MEASURE_CONDUCTIVITY
425       state.rfv [cnt1] = 0.;
426   #   endif
427     }
428     state.kinE = 0.0;
429     state.potE = 0.0;
430     state.virS = 0.0;


431     for (cntx = 0; cntx < boxes.numBoxXY; cntx++)
432       for (cnty = 0; cnty < boxes.numBoxXY; cnty++)
433         for (cntz = boxes.ziMin; cntz <= boxes.ziMax; cntz++)
434           for (cntc = 0; cntc < boxes.count[cntx] [cnty] [cntz]; cntc++) {
435             cnt1 = boxes.box[cntx] [cnty] [cntz] [cntc];

436             if (cnt1 < Mvnat)
437               continue;

438             if (state.ion [cnt1 ] < 0)
439               continue;


440             double prevPotE = state.potE;
441             vector<int> coulombNeighbours;
442             vector<int> swNeighbours;

443             if ( incl_Coulomb ) {
444               // prep. for Coulomb loop
445               if (Num_el + Numion > 1) {

446                 x3start = cntx - 1;
447                 x3stop  = cntx + 1;
448                 y3start = cnty - 1;
449                 y3stop  = cnty + 1;
450                 z3start = cntz - 1;
451                 z3stop  = cntz + 1;

452   #             if ! PERIODIC_BOUNDARIES_X
453                 if (x2start <              0) x2start = 0;
454                 if (x2stop  >= boxes.numBoxXY) x2stop  = boxes.numBoxXY - 1;
455   #             endif
456   #             if ! PERIODIC_BOUNDARIES_Y
457                 if (y2start <              0) y2start = 0;
```

```
458             if (y2stop  >= boxes.numBoxXY) y2stop  = boxes.numBoxXY - 1;
459 #           endif
460 #           if ! PERIODIC_BOUNDARIES_Z
461             if (z2start <              0) z2start = 0;
462             if (z2stop  >=  boxes.numBoxZ) z2stop  = boxes.numBoxZ - 1;
463 #           endif


464             // 2nd loop Coulomb
465             for (cntx2i = x2start; cntx2i <= x2stop; cntx2i++) {
466               cntx2 = cntx2i;
467 #             if PERIODIC_BOUNDARIES_X
468               if      (cntx2 <              0) cntx2 += boxes.numBoxXY;
469               else if (cntx2 >= boxes.numBoxXY) cntx2 -= boxes.numBoxXY;
470 #             endif
471               for (cnty2i = y2start; cnty2i <= y2stop; cnty2i++) {
472                 cnty2 = cnty2i;
473 #               if PERIODIC_BOUNDARIES_Y
474                 if      (cnty2 <              0) cnty2 += boxes.numBoxXY;
475                 else if (cnty2 >= boxes.numBoxXY) cnty2 -= boxes.numBoxXY;
476 #               endif
477                 for (cntz2i = z2start; cntz2i <= z2stop; cntz2i++) {
478                   cntz2 = cntz2i;
479 #                 if PERIODIC_BOUNDARIES_Z
480                   if      (cntz2 <              0) cntz2 += boxes.numBoxZ;
481                   else if (cntz2 >= boxes.numBoxZ) cntz2 -= boxes.numBoxZ;
482 #                 endif
483                   for (cntc2 = 0; cntc2 < boxes.count[cntx2] [cnty2] [cntz2]; cntc2++) {
484                     cnt2 = boxes.box[cntx2] [cnty2] [cntz2] [cntc2];

485                     if (state.ion [cnt2 ] < 0)
486                       continue;

487                     if (cnt1 == cnt2)
488                       continue;


489                     // Coulomb potential
490                     // ~~~~~~~~~~~~~~~~~~
491                     if (state.ion [cnt1 ] > 0)
492                       if (state.ion [cnt2 ] > 0) {

493                         xij =   state.x[cnt1] - state.x[cnt2];
494                         yij =   state.y[cnt1] - state.y[cnt2] ;
495                         zij =   state.z[cnt1] - state.z[cnt2] ;

496 #                       if PERIODIC_BOUNDARIES_X
497                         if      ( xij >  params.volHalfWidth ) xij -= params.volWidth;
498                         else if ( xij < -params.volHalfWidth ) xij += params.volWidth;
499 #                       endif
500 #                       if PERIODIC_BOUNDARIES_Y
501                         if      ( yij >  params.volHalfWidth ) yij -= params.volWidth;
```

```
502                         else if ( yij < -params.volHalfWidth ) yij += params.volWidth;
503 #                       endif
504 #                       if PERIODIC_BOUNDARIES_Z
505                         if      ( zij >  params.volHalfHeight ) zij -= params.volHeight;
506                         else if ( zij < -params.volHalfHeight ) zij += params.volHeight;
507 #                       endif

508                         rdsq = xij * xij + yij * yij + zij * zij;

509                         if (rdsq >= CLRCSQ)
510                           continue;

511                          fcoul = 1.0; // if attractive then fcoul negative
512                          if (cnt1 > NAT ) {
513                           fcoul *= -1.;
514                           ltint2 = cnt1 ;
515                           ltint1 = cnt2 ;
516                         }
517                         if (cnt2 > NAT ) {
518                           fcoul *= -1.;
519                           ltint1 = cnt1 ;
520                           ltint2 = cnt2 ;
521                         }

522                         rdsr = sqrt (rdsq);
523                         pot = fcoul * 0.14399762e2 * state.ion [cnt1 ] * state.ion [cnt2 ] / rdsr;
524                         fcoul *= pot / rdsq;

525                         if (fcoul < 0.0 && rdsq < recombine_radiusq ) // minimum radius for e-i+
526                           error += recombine ( ltint1 , ltint2 );
527                         else
528                           state.pot [cnt1] += pot;

529                         state.fx [cnt1] += xij * fcoul;
530                         state.fy [cnt1] += yij * fcoul;
531                         state.fz [cnt1] += zij * fcoul;


532                         vir = -fcoul * rdsq;
533                         state.virS  += vir;
534                         vir  *= 0.5;
535                         state.virial[cnt1] += vir;
536                         state.virial[cnt2] += vir;
537 #                       if MEASURE_CONDUCTIVITY
538                         fv = (xij * fcoul) * state.vx[cnt1]
539                           +  (yij * fcoul) * state.vy[cnt1]
540                           +  (zij * fcoul) * state.vz[cnt1];
541                         state.rfv[cnt1].x += (-xij)*fv;
542                         state.rfv[cnt1].y += (-yij)*fv;
543                         state.rfv[cnt1].z += (-zij)*fv;
544 #                       endif
545                       }
```

```
546                     coulombNeighbours.push_back(cnt2);
547                   }
548                 }
549               }
550             }
551           }
552         } // end of Coulomb potential

553         if (!sti_web) {
554           // SW interaction range is only one box length
555           x2start = cntx - 1;
556           x2stop  = cntx + 1;
557           y2start = cnty - 1;
558           y2stop  = cnty + 1;
559           z2start = cntz - 1;
560           z2stop = cntz + 1;

561 #         if ! PERIODIC_BOUNDARIES_X
562           if (x2start <              0) x2start = 0;
563           if (x2stop  >= boxes.numBoxXY) x2stop  = boxes.numBoxXY - 1;
564 #         endif
565 #         if ! PERIODIC_BOUNDARIES_Y
566           if (y2start <              0) y2start = 0;
567           if (y2stop  >= boxes.numBoxXY) y2stop  = boxes.numBoxXY - 1;
568 #         endif
569 #         if ! PERIODIC_BOUNDARIES_Z
570           if (z2start <              0) z2start = 0;
571           if (z2stop  >=  boxes.numBoxZ) z2stop  = boxes.numBoxZ - 1;
572 #         endif


573           for (cntx2i = x2start; cntx2i <= x2stop; cntx2i++) {
574             cntx2 = cntx2i;
575 #           if PERIODIC_BOUNDARIES_X
576             if      (cntx2 <              0) cntx2 += boxes.numBoxXY;
577             else if (cntx2 >= boxes.numBoxXY) cntx2 -= boxes.numBoxXY;
578 #           endif
579             for (cnty2i = y2start; cnty2i <= y2stop; cnty2i++) {
580               cnty2 = cnty2i;
581 #             if PERIODIC_BOUNDARIES_Y
582               if      (cnty2 <              0) cnty2 += boxes.numBoxXY;
583               else if (cnty2 >= boxes.numBoxXY) cnty2 -= boxes.numBoxXY;
584 #             endif
585               for (cntz2i = z2start; cntz2i <= z2stop; cntz2i++) {
586                 cntz2 = cntz2i;
587 #               if PERIODIC_BOUNDARIES_Z
588                 if      (cntz2 <              0) cntz2 += boxes.numBoxZ;
589                 else if (cntz2 >= boxes.numBoxZ) cntz2 -= boxes.numBoxZ;
590 #               endif

591                 for (cntc2 = 0; cntc2 < boxes.count [cntx2] [cnty2] [cntz2]; cntc2++) {
```

```
592                     cnt2 = boxes.box[cntx2] [cnty2] [cntz2] [cntc2];

593                     if (state.ion [cnt2 ] < 0)
594                       continue;

595                     if (cnt1 == cnt2)
596                       continue;

597                     if (cnt1 > NAT )
598                       continue;
599                     if (cnt2 > NAT )
600                       continue;
601                     if ((state.ion [cnt1 ] > 0) && (state.ion [cnt2 ] > 0))
602                       continue;

603                     xij =  state.x[cnt1 ] - state.x[cnt2 ];
604                     yij =  state.y[cnt1 ] - state.y[cnt2 ];
605                     zij =  state.z[cnt1 ] - state.z[cnt2 ];

606 #                   if PERIODIC_BOUNDARIES_X
607                     if      ( xij >  params.volHalfWidth ) xij -= params.volWidth;
608                     else if ( xij < -params.volHalfWidth ) xij += params.volWidth;
609 #                   endif
610 #                   if PERIODIC_BOUNDARIES_Y
611                     if      ( yij >  params.volHalfWidth ) yij -= params.volWidth;
612                     else if ( yij < -params.volHalfWidth ) yij += params.volWidth;
613 #                   endif
614 #                   if PERIODIC_BOUNDARIES_Z
615                     if      ( zij >  params.volHalfHeight ) zij -= params.volHeight;
616                     else if ( zij < -params.volHalfHeight ) zij += params.volHeight;
617 #                   endif

618                     rdsq = xij * xij + yij * yij + zij * zij;

619                     if  (rdsq < RCSQ) {
620                       // two body POTENTIAL
621                       //if ( true ) cerr << "warning: removed 2-body potential" << endl; else
622                       two_pot  (xij , yij , zij , cnt1 , cnt2 );

623                       // is bond broken ??             THREE BODY POTENTIAL
624                       tflt = (double) state.exc [cnt1 ];
625                       if (!incl_brkbnd || myRandom.rand() >= tflt / 4. ) {

626                         x3start = cntx - 1;
627                         x3stop  = cntx + 1;
628                         y3start = cnty - 1;
629                         y3stop  = cnty + 1;
630                         z3start = cntz - 1;
631                         z3stop  = cntz + 1;

632 #                       if ! PERIODIC_BOUNDARIES_X
633                         if (x3start <              0) x3start = 0;
```

```
634                         if (x3stop  >= boxes.numBoxXY) x3stop  = boxes.numBoxXY - 1;
635 #                       endif
636 #                       if ! PERIODIC_BOUNDARIES_Y
637                         if (y3start <               0) y3start = 0;
638                         if (y3stop  >= boxes.numBoxXY) y3stop  = boxes.numBoxXY - 1;
639 #                       endif
640 #                       if ! PERIODIC_BOUNDARIES_Z
641                         if (z3start <               0) z3start = 0;
642                         if (z3stop  >=  boxes.numBoxZ) z3stop  = boxes.numBoxZ - 1;
643 #                       endif

644                         //if ( cnt1==1 )
645                         three_pot  (xij, yij, zij, cnt1, cnt2);

646                       }
647                       //else cout << "broken bond: " << cnt1 << endl;

648                       swNeighbours.push_back(cnt2);
649                     }
650                   }
651                 }
652               }
653             }
654           }

655           if (fabs(state.potE-prevPotE) > 100.0) {
656             printf("warning: high-energy particle %5ld, box(%2d,%2d,%2d), ",
657                    cnt1,cntx,cnty,cntz);
658             printf("pos(%5.2f,%5.2f,%5.2f), potE=%.2f\n",
659                    state.x[cnt1],state.y[cnt1],state.z[cnt1], state.potE-prevPotE);
660           }
661 /*1*/
662         }


663   if (sti_web && incl_LJ) // only use LJ-pot if no SW-pot
664     error += potLJ(state,boxes);

665 # if BENCHMARK_FORCES
666   clock.stop();
667 # if 0
668   cout << "forces step took " << clock.elapsed() << " seconds" << endl;
669 # else
670   const char* filename_benchmark = "comptime_forces.dat";
671   static ofstream fout; //(filename_benchmark);
672   if (!(fout.is_open()) || !(fout.good())) {
673     extern int cont_prg;
674     if (cont_prg == 1 && fileExists(filename_benchmark)) {
675       fout.open(filename_benchmark,ios::app);
676     }
677     else {
678       fout.open(filename_benchmark);
```

```
679         fout << "#incl.el.\tsim.time\tcpu.time[s]" << endl;
680       }
681     }
682     fout << sti_web << "\t" << realtime << "\t" << clock.elapsed() << endl;
683   # endif
684   # endif

685     return error;
686   }
```

D.4.4 Heat Flow

HeatFlow.h

```
1   #ifndef _HeatFlow_h_
2   #define _HeatFlow_h_

3   #include "const.h"

4   #include <iostream>
5   #include <fstream>
6   #include <string>
7   #include <vector>

8   #include "physics.h"
9   #include "input.h"

10  #define HF_WITH_ELECTRONS 1
11  #define HF_WITH_ENERGY_BUFFER 0


12  #if HF_WITH_ELECTRONS
13  #undef HF_WITH_ENERGY_BUFFER
14  #define HF_WITH_ENERGY_BUFFER 0
15  #elif ! HF_WITH_ENERGY_BUFFER
16  #undef HF_WITH_ELECTRONS
17  #define HF_WITH_ELECTRONS 1
18  #endif


19  using namespace std;

20  class State;
21  class HeatFlow;

22  class HeatFlowSystem {
23    public:
```

```
24      HeatFlowSystem(const string& desc = "no-name");
25      virtual ~HeatFlowSystem() { }

26      void init();                    // initializes the system
27      bool step(double p1, double p2);        // integrates system to given time

28      double getTemperature(double z) const;   // returns the temperature
29      void setTemperature(double temperature, int cell, double time); // sets temperature
30      double getTime() const;                  // return current time
31      void setTime(double t);                 // set current time
32      double sumEnergy() const;                // return system energy in eV/A2
33      void setDensity(double d);       // sets the system to the given mass density

34      void printInfo(ostream& out) const;
35      void printTemperature(const string& filename = "", bool append=false);
36      void printEnergy(const string& filename = "", bool append=false);

37      inline double getEnergyOut() const { return energyOut; }
38      inline double getCellSize() const { return dz; }

39      virtual void save(ostream& out) const;
40      virtual void load(istream& in);

41      typedef vector<double> Array1D; // the data structure to use

42    protected:

43      Array1D density;    // mass density
44      Array1D temp;       // temerature
45      Array1D energy;     // energy density
46      Array1D extEnergy;  // external energy influx
47      Array1D extDensity; // external density influx
48      Array1D gTrans;     // energy transfer between systems (el-ph)
49      Array1D cond;       // heat conductivity
50      double dt;          // time steps (femtoseconds)
51      double dz;          // cell length (Angstrom)
52      int numCells;       // number of cells
53      double zMax;        // z-coordinate of interface between MD and this model

54      double time;        // current time
55      double energyOut;   // energy flowing out of the system
56      double sumExtE;     // sum up all energy collected from external sources
57      double densityInit; // initial mass density
58      double energyInit;  // initial energy density

59      // boundary conditions:
60      double tempInf;     // temperature at infinity

61      // limiting conditions
62      double maxTemp;     // simulation potentially unstable if temp. larger than this
63      double maxD;        // max. heat diffusivity
```

```
64      // file output streams
65      ofstream outTemp;
66      ofstream outEnergy;

67      string description; // a name to identify the system (electron or lattice)

68      virtual double heatCapacity(double temperature) const = 0;
69      void setNumCells(int num, bool initialize = true);
70      void setCellSize(double deltaZ, double timeStep);

71      friend class HeatFlow;
72  };



73  class HeatFlowSystemPh : public HeatFlowSystem {
74    public:

75      HeatFlowSystemPh(const string& desc = "no-name");
76      virtual ~HeatFlowSystemPh() { }

77      void printInfo(ostream& out) const;

78      virtual void save(ostream& out) const;
79      virtual void load(istream& in);

80    protected:

81      virtual inline double heatCapacity(double temperature) const {
82        return HF_HEATCAP * PhysConv::Joules_to_eV / PhysConv::g_to_rmu;
83      }

84      double meltTemp;      // melting temperature (lattice)

85      friend class HeatFlow;
86  };

87  #if HF_WITH_ELECTRONS

88  class HeatFlowSystemEl : public HeatFlowSystem {
89    public:

90      HeatFlowSystemEl(const string& desc = "no-name");
91      virtual ~HeatFlowSystemEl() { }

92      // void init();      // initializes the system
93      void printInfo(ostream& out) const;

94      virtual void save(ostream& out) const;
95      virtual void load(istream& in);
```

```
96    protected:

97      virtual inline double heatCapacity(double temperature) const {
98        //return factorHeatCap * temperature;
99        return 1.5 * PhysConstRU::K / PhysConstRU::mass_el;
100     }

101     double factorHeatCap; // scaling factor for computing heat cap.
102     friend class HeatFlow;
103 };

104 #endif

105 class HeatFlow {
106   public:

107     HeatFlow(bool verbose=true);

108     void printInfo(ostream& out = cout) const; // prints some information about the setup
109     void init(bool verbose=true); // initializes the system
110     bool step(double destTime);
111     bool step(); // integrates system by one time step
112     void laser(double laserIntensity, double duration); // intensity is in eV/(fs*A^2)
113     void setTemperatureTop(double temperature, double time); // sets boundary temperature (at top)

114     double getLaserEnergy() const; // returns input laser energy density in eV/A3
115     double getLaserFluence() const; // returns input laser fluence in J/cm2
116     double getTime() const; // return current time
117     void setTime(double t); // set current time
118     double sumEnergy() const; // return system energy in eV
119     double getMaxZ() const; // return the z-coordinate at top of model
120     double getMinZ() const; // return the z-coordinate at bottom of model
121     double getTimeStep() const; // return length of time step (in fs)
122     inline double getEnergyOut() const;

123     void dampParticle(State& state, long pid) const;

124     void save(ostream& out) const;
125     void load(istream& in);

126     static int test(); // runs heat-flow simulation for testing/debugging


127     HeatFlowSystemPh phSystem;
128 #   if HF_WITH_ELECTRONS
129     HeatFlowSystemEl elSystem;
130 #   elif HF_WITH_ENERGY_BUFFER
131     HeatFlowSystem::Array1D laserEnergy;
132     double gParam;      // coupling parameter
133     double minTransE;  // min. unit of energy transfer (to avoid rounding errors)
134 #   endif
```

```
135   private:

136     double dt;      // time steps of MD simulation (femtoseconds)
137     double time;    // current time
138     double zMax;    // z-coordinate of interface between MD and this model

139     // physical properties of lattice and electron gas:
140     double absCoeff;   // absorption coefficient (1/A)

141 #   if HF_WITH_ELECTRONS
142     double gParam;     // coupling parameter 'G'
143     double Eg;         // band gap energy
144 #   endif

145     // some other parameters or variables
146     double phK300;          // lattice heat capacity at T=300 [eV/(fs A K)]
147     double gamma;           // factor for Langeving damping

148     double totalLaserFluence; // sums up laser fluence collected
149     double totalLaserEnergy; // sums up laser energy (density) collected

150 #   if PRINT_ABS_PROFILE
151     ofstream foutAP;
152 #   endif
153 };

154 #endif
```

HeatFlow.cc

```
1   #include "const.h"
2   #include "HeatFlow.h"

3   #include <iostream>
4   #include <fstream>
5   #include <sstream>
6   #include <iomanip>
7   #include <exception>
8   #include <string>
9   #include <algorithm>
10  #include <cassert>
11  #include <cmath>

12  #include "physics.h"
13  #include "State.h"
14  #include "Boxes.h"
15  #include "Random.h"
16  #include "bath.h"
17  #include "utility.h"
```

```
18  #define DEBUG_HEATFLOW 0
19  #define USE_EXPLICIT 1

20  extern double TEMP;
21  extern double DEBYE;

22  HeatFlow::HeatFlow(bool verbose)
23    : phSystem("lattice")
24  #   if HF_WITH_ELECTRONS
25      ,elSystem("electrons")
26  #   endif
27  {
28    double tempInf = TEMP;

29    extern double DIST_DMP;
30    int numLayers = (int)(DIST_DMP/boxes.boxLength);
31    int ziStart = boxes.firstNonDamped;
32    int ziEnd = ziStart + numLayers;
33    int boxIndex = boxes.boxMinZ + ziEnd - boxes.ziMin;
34    zMax = boxes.lowZ + boxIndex * boxes.boxLength;

35    phSystem.tempInf = tempInf;
36    phSystem.zMax = zMax;

37  # if HF_WITH_ELECTRONS
38    elSystem.tempInf = tempInf;
39    elSystem.zMax = zMax;
40  # endif

41    init(verbose);
42    if (verbose)
43      printInfo();
44  }

45  extern double SPACE;
46  extern double ABS_COEF;
47  extern double ABS_COEF2;
48  extern double OVER_WEIGH;
49  extern double LEngDen;
50  extern double time_st[2];
51  extern double LPhotEng;
52  extern double DIST_DMP;
53  extern double LIntensity;
54  extern double SiLIFET;

55  void HeatFlow::init(bool verbose)
56  {
57    time = 0;

58    phSystem.init();
59    phSystem.time = time;
```

```
60  # if HF_WITH_ELECTRONS
61    elSystem.init();
62    elSystem.time = time;
63  # endif

64    //cout << "HeatFlow::init(): time step" << endl;
65    dt = time_st[0]; // time step (atoms)
66    if (verbose)
67      cout << "time step = " << dt << " fs" << endl;

68    // Diamond lattice: 8 atoms per unit cell
69    phSystem.densityInit = 8*state.massAt / (SPACE*SPACE*SPACE); // eV*fs^2/A^5


70  # if HF_WITH_ELECTRONS
71    Eg = 1.11; // eV, for Silicon at 300K (Kittel, pg 201)

72    double phTemp = phSystem.tempInf;
73    //double phTemp = 1000;
74    double elNDensity =
75      2 * pow( state.massEl * PhysConstRU::K * phTemp
76               / (2 * Math::PI * PhysConstRU::hbar * PhysConstRU::hbar), 1.5 )
77      * exp(-Eg/(2 * PhysConstRU::K * phTemp)); // in A^-3

78    //elNDensity = 1.0275e+28 * (PhysConv::A_to_m*PhysConv::A_to_m*PhysConv::A_to_m);
79    elSystem.densityInit = elNDensity * state.massEl;// mass density
80    //elSystem.densityInit = 4*phSystem.densityInit/state.massAt * state.massEl;


81  # if 1
82    const double hbar2RU = PhysConstRU::hbar * PhysConstRU::hbar;
83    const double PI2 = Math::PI * Math::PI;
84    double fermiE = hbar2RU/(2*state.massEl) * pow( 3 * PI2 * elNDensity, 2.0/3.0 );
85    double fermiT = fermiE / PhysConstRU::K;
86    double elC = 0.5 * PI2 * PhysConstRU::K / (fermiT * state.massEl);

87    if (verbose) {
88      cout << "elNDensity        = " << elNDensity << " 1/A3" << endl;
89      cout << "fermi energy      = " << fermiE << " eV" << endl;
90      cout << "fermi temperature = " << fermiT << " K" << endl;
91      cout << "C_el-factor       = " << elC << " eV/(rmu*T2)" << endl;
92      cout << "C_el (at 300K)    = " << elC*300 << " eV/(rmu*T)" << endl;
93    }
94  # endif

95    elSystem.factorHeatCap =
96      (PhysConstRU::K*PhysConstRU::K) / (PhysConstRU::hbar*PhysConstRU::hbar)
97      * state.massEl * pow(elNDensity * Math::PI*Math::PI/9, 1.0/3.0)
98      / (PhysConstRU::mass_el*elNDensity);

99    //elSystem.factorHeatCap = 0.0;
```

```
100  # endif

101    if (verbose) { // debugging
102      double phHeatCap = phSystem.heatCapacity(phSystem.tempInf);
103      cout << "lattice K/(eV/A3): " << 1.0/(phHeatCap*phSystem.densityInit) << endl;
104  #   if HF_WITH_ELECTRONS
105      double elHeatCap = elSystem.heatCapacity(elSystem.tempInf);
106      cout << "el.-gas K/(eV/A3): " << 1.0/(elHeatCap*elSystem.densityInit) << endl;
107  #   endif
108    }


109  # if 0
110    double conv_A3_to_cm3 = PhysConv::A_to_cm * PhysConv::A_to_cm * PhysConv::A_to_cm;
111    cout << "PhysConv::eV_to_Joules = " << PhysConv::eV_to_Joules << endl;
112    cout << "PhysConv::amu_to_rmu   = " << PhysConv::amu_to_rmu << endl;
113    cout << "Math::PI               = " << Math::PI << endl;
114    cout << "PhysConst::hbar        = " << PhysConst::hbar << endl;
115    cout << "PhysConst::K           = " << PhysConst::K << endl;
116  # if HF_WITH_ELECTRONS
117    cout << "elDensity              = " << elSystem.Density << " eV*fs^2/A^5" << endl;
118    cout << "el. #Density           = " << elNDensity << " A^-3" << endl;
119    cout << "el. #Density           = " << elNDensity / conv_A3_to_cm3 << " cm^-3" << endl;
120    cout << "state.massEl           = " << state.massEl << " eV*fs^2/A^2" << endl;
121    cout << "Eg                     = " << Eg << " eV" << endl;
122  # endif
123    cout << "phTempInf              = " << phSystem.tempInf << " K" << endl;
124    cout << "PhysConstRU::K         = " << PhysConstRU::K << endl;
125    cout << "PhysConstRU::hbar      = " << PhysConstRU::hbar << endl;
126  # endif



127    double absCoeff1 = ABS_COEF*PhysConv::A_to_cm * OVER_WEIGH;
128    double absCoeff2 = ABS_COEF2*1e-9 * LIntensity * PhysConv::A_to_cm * OVER_WEIGH;
129    absCoeff = absCoeff1 + absCoeff2;

130    double skinDepthLin = 1. / absCoeff;
131    double simHeight = 5. * skinDepthLin; // desired height of simulation volume

132    // estimate maximum temperature
133    phSystem.meltTemp = 1687; // melt temperature for silicon (Kelvin)

134    //double laserEnergyDensity = LEngDen *
135    //  PhysConv::Joules_to_eV*(PhysConv::A_to_cm*PhysConv::A_to_cm);
136    //elSystem.maxTemp = sqrt(elSystem.tempInf*elSystem.tempInf +
137    //                        2*absCoeff*laserEnergyDensity
138    //                        / ( elSystem.factorHeatCap * elSystem.densityInit ) );
139  # if HF_WITH_ELECTRONS
140    elSystem.maxTemp = 2.0/(3*PhysConstRU::K) * ( LPhotEng - Eg ) * 5.0;
141  # endif
142    phSystem.maxTemp = 5*phSystem.meltTemp;
```

```
143   { // computing max. heat diffusivity

144     // for silicon: maxD = 0.85 cm2/s (Bauerle, pg.697)
145     double phCond = 1.5; // W/(cm K), conductivity for Si at T=300K
146     phCond *= (PhysConv::Joules_to_eV/PhysConv::s_to_fs) / PhysConv::cm_to_A; // eV/(fs A K)


147     // computing max. heat diffusivity for lattice
148     double minHeatCapPh = phSystem.heatCapacity(phSystem.tempInf);
149     double maxCondPh = phCond * phSystem.maxTemp / phSystem.tempInf;
150     phSystem.maxD = maxCondPh / (phSystem.densityInit*minHeatCapPh);

151 #   if HF_WITH_ELECTRONS
152     // computing max. heat diffusivity for electrons
153     double laserFluence = LEngDen; // J/cm2
154     laserFluence *= PhysConv::Joules_to_eV*PhysConv::A_to_cm*PhysConv::A_to_cm; // eV/A^2
155     double maxDensityEl = laserFluence * absCoeff;
156     double maxCond1El = phCond * elSystem.maxTemp / phSystem.tempInf;
157     double maxCond2El = phCond * elSystem.tempInf / phSystem.tempInf;

158     double d1 = maxCond1El / ( maxDensityEl * elSystem.heatCapacity(elSystem.maxTemp) );
159     double d2 = maxCond2El / ( elSystem.densityInit * elSystem.heatCapacity(elSystem.tempInf) );

160     //cout << "maxD (1) = " << d1 << endl;
161     //cout << "maxD (2) = " << d2 << endl;

162     elSystem.maxD = max( d1, d2 );
163     //elSystem.maxD = d1;

164     double physMaxD = PhysConst::c*PhysConv::m_to_A*PhysConv::fs_to_s * 0.1;
165     physMaxD *= physMaxD;
166     if ( elSystem.maxD > physMaxD )
167       elSystem.maxD = physMaxD;
168 #   endif
169   }

170   //phSystem.setCellSize(100., dt);
171   phSystem.setCellSize(DIST_DMP, dt);
172 # if HF_WITH_ELECTRONS
173   elSystem.setCellSize(2.0 * phSystem.dz, phSystem.dt);
174 # endif

175   {
176     // compute initial energy density:
177     const int n = 200;

178     const double dTPh = phSystem.tempInf / n;
179     double tPh = 0.5 * dTPh;

180 #   if HF_WITH_ELECTRONS
```

```
181       const double dTEl = phSystem.tempInf / n;
182       double tEl = 0.5 * dTEl;
183   #   endif

184       double sumPh = 0.0;
185   #   if HF_WITH_ELECTRONS
186       double sumEl = elNDensity * Eg;
187   #   endif
188       for ( int i = 0; i < n; ++i ) {
189         sumPh += phSystem.densityInit * phSystem.heatCapacity(tPh) * dTPh;
190         tPh += dTPh;
191   #     if HF_WITH_ELECTRONS
192         sumEl += elSystem.densityInit * elSystem.heatCapacity(tEl) * dTEl;
193         tEl += dTEl;
194   #     endif
195       }
196       phSystem.energyInit = sumPh;
197   #   if HF_WITH_ELECTRONS
198       elSystem.energyInit = sumEl;
199   #   endif
200     }



201     // create required number of cells
202     //cout << "HeatFlow::init(): create required number of cells" << endl;
203   # if HF_WITH_ELECTRONS
204     int numElCells = (int)ceil(simHeight / elSystem.dz);
205     if (numElCells < 10) numElCells = 10;
206     elSystem.setNumCells(numElCells);
207     simHeight = (elSystem.numCells-2) * elSystem.dz;
208     phSystem.setNumCells((int)(simHeight/phSystem.dz+0.5));
209   # else
210     int numPhCells = (int)ceil(simHeight / phSystem.dz);
211     phSystem.setNumCells(numPhCells);
212     simHeight = (phSystem.numCells-2) * phSystem.dz;
213   # endif

214     phSystem.setDensity( phSystem.densityInit ); // eV*fs^2/A^5
215   # if HF_WITH_ELECTRONS
216     elSystem.setDensity( elSystem.densityInit ); // mass density
217   # endif

218     totalLaserFluence = 0.0;
219     totalLaserEnergy = 0.0;

220   # if HF_WITH_ELECTRONS
221     //gParam = 1e16; // W/(m^3*K) - Norris et al., RSI, vol. 74, pg. 400 (2003) - Metal
222     gParam = 0.4e17; // W/(m^3*K) - assumes n_el ~ 1e21 cm^(-3)
223     gParam *= PhysConv::Joules_to_eV*PhysConv::fs_to_s
224       * PhysConv::A_to_m*PhysConv::A_to_m*PhysConv::A_to_m;
225   # endif
```

```
226   // gamma = pi/6 * (debye frequency)
227   // (debye frequency) = wD = kB/hbar * (debye temperature)
228   // see Kittel, Intro to Solid State Physics, 7th ed., pg. 122
229   // and Haberland et. al, PRB, yr. 1995, vol. 51, pg. 11061
230   gamma = - Math::PI/6.0*PhysConstRU::K/PhysConstRU::hbar*DEBYE; // [1/fs]
231   if (gamma < -1.)
232     gamma = -1.0;

233 # if HF_WITH_ENERGY_BUFFER
234   gParam = 1.0-exp(-phSystem.dt/SiLIFET);
235   minTransE = 1.0e-4 * phSystem.energyInit;
236   laserEnergy.resize(phSystem.energy.size());
237   for (unsigned int i = 0; i < laserEnergy.size(); ++i)
238     laserEnergy[i] = 0.0;
239 # endif
240 } // end of HeatFlow::init()



241 // adds laser energy to the electron gas.  intensity is in eV/(fs*A^2)
242 void HeatFlow::laser(double laserIntensity, double duration)
243 {
244 # if HF_WITH_ELECTRONS
245   HeatFlowSystem& hfSystem = elSystem;
246 # else
247   HeatFlowSystem& hfSystem = phSystem;
248 # endif

249   //cout << "laserIntensity = " << laserIntensity << endl;

250   double absCoeff1 = ABS_COEF*PhysConv::A_to_cm * OVER_WEIGH;
251   double absCoeff2 = ABS_COEF2*1e14*PhysConv::eV_to_Joules * laserIntensity * OVER_WEIGH;
252   absCoeff = absCoeff1+absCoeff2;

253   double factor = duration * laserIntensity * absCoeff;
254   double sumEnergy = 0.0;

255 # if PRINT_ABS_PROFILE
256   vector<double> nPhotAbs(hfSystem.numCells-2);
257   for ( unsigned int ii = 0; ii < nPhotAbs.size(); ++ii )
258     nPhotAbs[ii] = 0;
259   const double dz = hfSystem.dz;
260   const double dA = params.volWidth*params.volWidth;
261   const double dV = dz*dA;
262 # endif

263   for ( int i = 1; i < hfSystem.numCells-1; ++i ) {
264     double z = (i-0.5)*hfSystem.dz;
265     double energy = factor * exp(-absCoeff * z);

266     double nPhotons = energy / LPhotEng;
```

```
267  #   if HF_WITH_ELECTRONS
268      double incDensity = nPhotons * state.massEl; // increase in electron density
269      double incEnergy = nPhotons * ( LPhotEng - Eg );
270  #   else
271      double incEnergy = nPhotons * ( LPhotEng );
272  #   endif

273  #   if PRINT_ABS_PROFILE
274      nPhotAbs[i-1] = nPhotons * dV;
275  #   endif


276  #   if 0
277      if ( i % 1000 == 1 )
278        cout << "energy(z=" << setw(8) << z << " A)"
279             << " = " << energy << " eV/A3" << endl;
280  #   endif


281  #   if HF_WITH_ENERGY_BUFFER
282      laserEnergy[i] += incEnergy;
283  #   else
284      hfSystem.extEnergy[i] += incEnergy;
285  #   if HF_WITH_ELECTRONS
286      hfSystem.extDensity[i] += incDensity;
287  #   endif
288  #   endif

289  #   if 0
290      {
291        double temp = incEnergy / ( incDensity * hfSystem.heatCapacity(hfSystem.temp[i]) );
292        cout << "laser-generated electron gas: cell " << setw(3) << i << ": "
293             << "T=" << temp << "K, "
294             << "E=" << incEnergy << "eV/A3, "
295  #          if HF_WITH_ELECTRONS
296             << "D=" << incDensity << "1/A3"
297  #          endif
298             << endl;
299      }
300  #   endif

301      sumEnergy += incEnergy;
302    }
303    totalLaserEnergy += sumEnergy;
304    totalLaserFluence += sumEnergy * hfSystem.dz;

305  # if PRINT_ABS_PROFILE
306    extern double LAS_ON;
307    extern double TIMEST;
308    //extern double LPULSLEN;
309    extern double realtime;
```

```
310   if ( !foutAP || LAS_ON-TIMEST <= realtime && realtime <= LAS_ON+TIMEST ) {
311     if (foutAP.is_open())
312       foutAP.close();
313     ostringstream fname;
314     extern int pulseCount;
315     extern double LWAVLEN;
316     fname << "laser.absprofile_" << pulseCount << ".hf" << flush;
317     cout << "printing absorption profile to '" << fname.str() << "'" << endl;
318     foutAP.open(fname.str().c_str());
319     foutAP << "# dz = " << dz << " A" << endl
320            << "# dA = " << dA << " A^2" << endl
321            << "# photonEnergy = " << LPhotEng << " eV" << endl
322            << "# wavelength = " << LWAVLEN << " nm" << endl;
323   }
324   foutAP << realtime;
325   for ( unsigned int ii = nPhotAbs.size()-1; ii > 0; --ii ) {
326     foutAP << "\t" << nPhotAbs[ii];
327   }
328   foutAP << endl;
329 # endif

330 # if 0
331   using namespace PhysConv;
332   cout << "intensity: " << laserIntensity << " eV/(fs*A^2) = "
333        << laserIntensity * eV_to_Joules*cm_to_A*cm_to_A*s_to_fs << " W/cm2 -- "
334        << "laser fluence: " << totalLaserFluence << " eV/A2 = "
335        << totalLaserFluence * eV_to_Joules*cm_to_A*cm_to_A
336        << " J/cm2"
337        << " -- peak energy added = " << duration * laserIntensity * absCoeff << " eV/A3"
338        << " -> dT(el) = "
339        << (duration * laserIntensity * absCoeff /
340            (elSystem.density*elSystem.factorHeatCap*elSystem.tempInf)) << " K"
341        << endl;
342 # endif
343 }

344 bool HeatFlow::step()
345 {
346   //cout << "HeatFlow::step():  begin" << endl;

347   static long iterationNum = 0;

348   bool ok = true;
349   const double errorToleranceTime = 1e-6;

350   int nPhSteps = (int)ceil(dt/phSystem.dt);
351   assert( fabs(nPhSteps*phSystem.dt-dt) < errorToleranceTime );

352 # if HF_WITH_ELECTRONS
353   assert( phSystem.numCells >= elSystem.numCells );

354   double nElStepsD = phSystem.dt/elSystem.dt;
```

```
355   assert( nElStepsD < INT_MAX/2 );
356   int nElSteps = (int)ceil(nElStepsD);

357   assert( fabs(nElSteps*elSystem.dt-phSystem.dt) < errorToleranceTime );

358   int totalElSteps = nElSteps*nPhSteps;
359   int nPhpEl = (phSystem.numCells-2)/(elSystem.numCells-2); // # of ph-cells per el-cell

360   int elCnt = totalElSteps;
361 # endif

362   int phCnt = nPhSteps;


363   for ( int tiPh = nPhSteps; tiPh > 0; --tiPh ) {
364     ++iterationNum;

365 #   if HF_WITH_ELECTRONS
366     // e-h recombination:
367     for ( int i = 1; i < elSystem.numCells-1; ++i ) {
368       double dD = (elSystem.densityInit-elSystem.density[i])*phSystem.dt/SiLIFET;
369       double dE = -(dD/state.massEl)*Eg/nPhpEl;
370       for ( int j = (i-1)*nPhpEl+1; j <= i*nPhpEl; ++j )
371         phSystem.gTrans[j] += dE;
372       elSystem.density[i] += dD;
373     }
374 #   endif

375 #   if HF_WITH_ENERGY_BUFFER
376     for (int i = 1; i < phSystem.numCells-1; ++i) {
377       if ( laserEnergy[i] > minTransE ) {
378         double dE = gParam * laserEnergy[i];
379         phSystem.gTrans[i] += dE;
380         laserEnergy[i] -= dE;
381       }
382       else {
383         phSystem.gTrans[i] += laserEnergy[i];
384         laserEnergy[i] = 0.0;
385       }
386     }
387 #   endif

388     double phK300 = 1.5; // W/(cm K), at T=300K
389     phK300 *= (PhysConv::Joules_to_eV*PhysConv::fs_to_s) / PhysConv::cm_to_A; // eV/(fs A K)

390     // initializing lattice heat conductivity
391     //cout << "HeatFlow::step():  initializing lattice heat conductivity" << endl;
392     for ( int i = 1; i < phSystem.numCells-1 && ok; ++i ) { // will later be temperature dependent
393       phSystem.cond[i] = phK300;


394 #     if HF_WITH_ELECTRONS
```

```
395         int iPh = i;
396         int iEl = (i-1)/nPhpEl+1;

397  #      if 1
398         double n = elSystem.density[iEl]/PhysConstRU::mass_el;
399         gParam = 3*n*PhysConstRU::K / SiLIFET;
400         double dTelph = elSystem.temp[iEl] - phSystem.temp[iPh];
401         double g = gParam * dTelph * phSystem.dt;
402  #      else
403         double phT0 = phSystem.temp[iPh];
404         double phC = phSystem.density[iPh] * phSystem.heatCapacity(phT0);

405         double elT0 = elSystem.temp[iEl];
406         double elC = elSystem.density[iEl] * elSystem.heatCapacity(elT0);

407         double Tf = (elC*elT0 + phC*phT0 ) / (elC+phC);
408         double beta = gParam * ( 1.0/elC + 1.0/phC );
409         double gPh =   phC * (Tf-phT0) * (1-exp(-beta*phSystem.dt));
410         double gEl = - elC * (Tf-elT0) * (1-exp(-beta*phSystem.dt));
411         assert( fabs(gPh-gEl) < 1e-6 );
412         double g = gEl;
413  #      endif

414         elSystem.gTrans[iEl] -= g;
415         phSystem.gTrans[iPh] += g;

416  #      if 0
417         cout << time << "fs -- " << iterationNum << ": " << iPh << "/" << iEl << ": "
418              << "Tf=" << Tf << ", transfering " << g << " eV/A3 "
419              << "from electrons (" << elT0 << "K, " << elC << "eV/A3K) "
420              << "to lattice (" << phT0 << ", " << phC << "eV/A3K)"
421              << endl;
422  #      endif

423  #      endif
424       }



425  #    if HF_WITH_ELECTRONS

426       for ( int iEl = nElSteps; iEl > 0 && ok; --iEl ) {

427         // initializing electron heat conductivity
428         //cout << "HeatFlow::step(): initializing electron heat conductivity" << endl;

429         elSystem.cond.front() =
430           phSystem.cond.front() * elSystem.temp.front()/phSystem.temp.front();
431         elSystem.cond.back() =
432           phSystem.cond.back() * elSystem.temp.back()/phSystem.temp.back();
433         for ( int i = 1; i < elSystem.numCells-1; ++i ) {
```

```
434         double sumInvCond = 0.0;
435         for ( int j = (i-1)*nPhpEl+1; j <= i*nPhpEl; ++j )
436           sumInvCond += phSystem.temp[i]/(elSystem.temp[i]*phSystem.cond[i]);
437         elSystem.cond[i] = nPhpEl/sumInvCond;
438       }

439       ok = ok && elSystem.step(1.0/elCnt, 1.0/iEl);
440       --elCnt;

441 #     if 0
442       if ( iEl%100 == 0) {
443         double sumE = 0.0;
444         for ( int i = 1; i < elSystem.numCells-1; ++i ) sumE+=elSystem.energy[i];
445         cout << "t = " << setw(10) << elSystem.time
446              << "   T = " << setw(10) << elSystem.temp[0]
447              << "   E = " << setw(10) << sumE * elSystem.dz << " eV/A2"
448              << "   extE = " << setw(10) << elSystem.sumExtE * elSystem.dz << " eV/A2"
449              << " = " << setw(10) << (elSystem.sumExtE*elSystem.dz*
450                                       PhysConv::eV_to_Joules*1e16) << " J/cm2"
451              << endl;
452       }
453 #     endif

454     }

455 #   endif // end of electron steps

456     ok = ok && phSystem.step(1.0/phCnt, 1.0/tiPh);
457     --phCnt;
458   }

459   time += dt;

460   if ( !ok ) return ok;

461   //cout << "HeatFlow::step():  consistency check" << endl;
462   assert( fabs(time-phSystem.time) < errorToleranceTime );
463   assert( phCnt == 0 );
464 # if HF_WITH_ELECTRONS
465   assert( fabs(time-elSystem.time) < errorToleranceTime );
466   assert( elCnt == 0 );
467 # endif
468   //cout << "HeatFlow::step():  end" << endl;

469   return true;
470 }

471 #if USE_EXPLICIT

472 bool HeatFlowSystem::step(double p1, double p2)
473 {
474   time += dt;
```

```
475   double dz2 = dz*dz;

476   Array1D dE(numCells);
477   Array1D dD(numCells);

478   for ( int i = 1; i < numCells-1; ++i ) {

479     // Enthalpy Equation: compute change in energy for each cell
480     double condP = 2.0 / (1.0/cond[i+1] + 1.0/cond[i]);
481     double condM = 2.0 / (1.0/cond[i] + 1.0/cond[i-1]);
482     double dTp = temp[i+1] - temp[i];
483     double dTm = temp[i] - temp[i-1];
484     double gTransE = p2*gTrans[i];
485     double extE = p1*extEnergy[i];
486     double extD = p1*extDensity[i];

487     gTrans[i] -= gTransE;
488     extEnergy[i] -= extE;
489     extDensity[i] -= extD;

490     sumExtE += extE;

491     dD[i] = extD;

492     dE[i] = dt/dz2 * ( condP*dTp - condM*dTm );
493     dE[i] += extE;
494     dE[i] += gTransE;

495     if ( i == numCells-2 ) {
496       energyOut += -dt/dz2 * condP*dTp;
497     }
498   }

499   for ( int i = 1; i < numCells-1; ++i ) {
500     double heatCap = heatCapacity(temp[i]);

501     // update energies
502     energy[i] += dE[i];

503     // Temperature eqn T=T(H)
504     temp[i] += dE[i]/(density[i]*heatCap);
505     temp[i] *= density[i] / (density[i]+dD[i]);

506     // update density
507     density[i] += dD[i];


508     if ( temp[i] > maxTemp ) {
509       using namespace PhysConv;
510       cerr << "HeatFlow::step(): " << description << " temperature (" << temp[i] << " K) "
511            << "has exceeded max. temperature (" << maxTemp << " K)"
```

```
512             << " -- cell# " << i << " at depth " << (i-0.5)*dz << " A"
513             << endl;
514       return false;
515     }
516     else if ( temp[i] <= 0 ) {
517       using namespace PhysConv;
518       cerr << "HeatFlow::step(): " << description << " temperature (" << temp[i] << " K) "
519             << " is out of bounds"
520             << " -- cell# " << i << " at depth " << (i-0.5)*dz << " A"
521             << endl;
522       return false;
523     }
524   }

525   // insulating boundary condition (heat transfer between MD and this model
526   // is done using phExtEnergy and elExtEnergy)
527   temp[0]     = temp[1];
528   temp.back() = tempInf;
529   energy[0]     = energy[1];
530   energy.back() = energyInit;

531   //cout << "HeatFlow::step(): t=" << time << " fs "
532   //     << " [" << description << "] T(top) = " << temp[0] << " K" << endl;

533   //printTemperature();
534   //printEnergy();
535   return true;
536 }


537 #else // USE implicit


538 bool HeatFlowSystem::step(double p1, double p2)
539 {
540   const double theta = 0.5;

541   time += dt;

542   const double dtdz = dt/dz;
543   const double tdtdz = theta * dtdz;
544   const double mtdtdz = (1-theta) * dtdz;

545   Array1D origE(numCells);
546   Array1D origT(numCells);
547   Array1D invHeatCap(numCells); // inverse of heat capacity for each cell
548   Array1D invResist(numCells-1); // inverse of thermal resistivity between cells i and i+1
549   vector<bool> thermalize(numCells);

550   double diffuseError = false;

551   for ( int i = 0; i < numCells-1; ++i ) {
```

```
552     if ( temp[i] > maxTemp ) {
553       using namespace PhysConv;
554       cerr << "HeatFlow::step(): " << description << " temperature (" << temp[i] << " K) "
555            << "has exceeded max. temperature (" << maxTemp << " K)"
556            << " -- cell# " << i << " at depth " << (i-0.5)*dz << " A"
557            << endl;
558       printTemperature();
559       printEnergy();
560       return false;
561     }
562     else if ( temp[i] <= 0 ) {
563       using namespace PhysConv;
564       cerr << "HeatFlow::step(): " << description << " temperature (" << temp[i] << " K) "
565            << " is out of bounds"
566            << " -- cell# " << i << " at depth " << (i-0.5)*dz << " A"
567            << endl;
568       return false;
569     }


570     double gTransE = p2*gTrans[i];
571     double extE = p1*extEnergy[i];
572     double extD = p1*extDensity[i];

573     gTrans[i] -= gTransE;
574     extEnergy[i] -= extE;
575     extDensity[i] -= extD;

576     origT[i] = temp[i] * density[i] / (density[i]+extD);
577     density[i] += extD;
578     invHeatCap[i] = 1.0/(density[i] * heatCapacity(temp[i])); // assume slowly varying heat capacity
579     origE[i] = energy[i] + extE + gTransE;
580     origT[i] += (origE[i]-energy[i]) * invHeatCap[i];

581     energy[i] = origE[i];
582     temp[i] = origT[i];

583     invResist[i] = 1.0 / ( 0.5 * dz * (1.0/cond[i+1] + 1.0/cond[i]) );


584     thermalize[i] = ( cond[i]*invHeatCap[i] > maxD );
585     diffuseError = diffuseError || thermalize[i];
586   }

587   invHeatCap.back() = 1.0/(density.back() * heatCapacity(temp.back()));

588   // adjust boundary condition (energy from MD is coupled into cell 1)
589   energy[0] = energy[1];
590   temp[0] = temp[1];

591   origE[0] = energy[0];
```

```
592   origE.back() = energy.back();

593   origT[0] = temp[0];
594   origT.back() = temp.back();

595   invResist[0] = 1.0 / ( 0.5 * dz * (1.0/cond[1]) );
596   invResist[numCells-2] = 1.0 / ( 0.5 * dz * (1.0/cond[numCells-2]) );

597   // thermalize regions where diffusivity is above threshold
598   if ( diffuseError ) {
599     //cerr << "diffusivity too large" << endl;

600     int begin = -1;
601     int end = begin;
602     while ( begin < (int)thermalize.size()-1 ) {
603       thermalize.back() = true;
604       while ( !thermalize[++begin] );
605       end = begin-1;
606       thermalize.back() = false;
607       while ( thermalize[++end] );

608       if ( begin < end ) {
609         int s = (begin>0?begin-1:0);
610         int e = (end<(int)thermalize.size()?end+1:thermalize.size());
611         double sumHeat = 0.0;
612         double sumHeatCap = 0.0;
613         for ( int i = s; i < e; ++i ) {
614           if ( invHeatCap[i] <= 0.0 )
615             cerr << "invHeatCap["<<i<<"] = " << invHeatCap[i] << endl;

616           double hc = 1.0/invHeatCap[i];
617           sumHeatCap += hc;
618           sumHeat += temp[i]*hc;
619         }
620         double newT = sumHeat/sumHeatCap;
621         for ( int i = s; i < e; ++i ) {
622           energy[i] += (newT-temp[i])/invHeatCap[i];
623           temp[i] = newT;
624         }
625       }

626       begin = end;
627     }
628   }

629   Array1D b(numCells-1);
630   Array1D xi(numCells-1);
631   for ( int i = 1; i < numCells-1; ++i ) {
632     b[i] = energy[i] + mtdtdz *
633       ( origT[i+1] * invResist[i]
634         - (invResist[i]+invResist[i-1]) * origT[i]
635         + origT[i-1]*invResist[i-1] );
```

```
636     xi[i] = tdtdz * (invResist[i] + invResist[i-1]);
637   }


638   const double tolerance = 1e-6;
639   double temperatureError = 2*tolerance;
640   int numIter = 0;
641   bool converged = false;

642   Array1D prevTemperature(numCells);
643   for ( int i = 0; i < numCells; ++i )
644     prevTemperature[i] = temp[i];

645   while ( !converged && temperatureError > tolerance ) {
646     ++numIter;

647     if ( numIter % 10000 == 0 ) cout << "time=" << time << ", iteration " << numIter << endl;

648     temperatureError = 0.0;
649     converged = true;
650     for ( int i = 1; i < numCells-1; ++i ) {
651       double currTemp = temp[i];
652       double phi = b[i] + tdtdz * (temp[i+1]*invResist[i] + temp[i-1]*invResist[i-1]);
653       energy[i] = ( phi + xi[i]*(origE[i]*invHeatCap[i] - origT[i]) ) / ( 1.0+xi[i]*invHeatCap[i] );
654       temp[i] = origT[i] + (energy[i]-origE[i])*invHeatCap[i];

655       double currError = fabs((temp[i]-currTemp)/currTemp);
656       temperatureError = max(temperatureError, currError);

657       double d1 = currTemp - prevTemperature[i];
658       double d2 = temp[i] - currTemp;
659       converged = converged && (currError <= tolerance) && (fabs(d2)<=fabs(d1));

660       prevTemperature[i] = currTemp;

661       if ( temp[i] > maxTemp || temp[i] < -maxTemp ) {
662         temperatureError = 0;
663         break;
664       }
665     }

666     energy[0] = energy[1];
667     temp[0] = temp[1];

668     //printTemperature();
669     //printEnergy();
670   }

671   // insulating boundary condition (heat transfer between MD and this model
672   // is done using phExtEnergy and elExtEnergy)
673   temp[0]     = temp[1];
```

```
674   energy[0]   = energy[1];

675   temp.back() = tempInf;
676   energy.back() = energyInit;

677   //printTemperature();
678   //printEnergy();

679   return true;
680 }

681 #endif


682 void HeatFlow::dampParticle(State& state, long pid) const
683 {
684   double z = state.z[pid];
685   if ( state.z[pid] > zMax ) z = zMax; // avoid the warning message
686   double temperature = phSystem.getTemperature(z);
687   ::dampParticle(state, pid, temperature);
688 }
```

D.4.5 Laser Absorption

laser_on.cc

```
1  /** \file laser_on.c
2      \author R.Herrmann, J.Gerlach, R.Holenstein
3      \date   7.10.96

4      \brief subroutines for laser simulation

5  */

6  #include "const.h"

7  #include "laser_on.h"
8  #include "laser.h"

9  #include <math.h>
10 #include <stdio.h>
11 #include <iostream>
12 #include <fstream>
13 #include <sstream>
14 #include <iomanip>
```

```
15  using namespace std;

16  #include "Random.h"
17  #include "Boxes.h"
18  #include "State.h"
19  #include "input.h"
20  #include "silicon.h"
21  #include "init.h"
22  #include "output.h"
23  #include "elec_fct.h"
24  #include "calc_par.h"
25  #include "utility.h"
26  #include "Array2D.h"
27  #include "HeatFlow.h"

28  #if SIM_WITH_MPI
29  #include "Communicator.h"
30  extern Communicator* comPtr;
31  #endif

32  extern HeatFlow* hfPtr;

33  // notes:
34  // ------
35  // possible optimization:
36  //   use photonCount from previous time step for current in order
37  //   for this function to properly parallelize (currently this function
38  //   is effectively executed in series, as each node has to wait for the
39  //   left over photons to be passed on from node above (except for top node
40  //   of course)
41  //

42  int pulseCount = 0;

43  /**
44     \brief check if atom gets excited during laser radiation
45     \param real_time timestep
46  */
47  void laser_on (double real_time )
48  {
49    static double totalLaserEnergy = 0.0;
50    static double totalAbsorbedEnergy = 0.0;
51    static ofstream fout;
52  # if PRINT_ABS_PROFILE
53    static ofstream foutAP;
54  # endif

55    const double intensity_unit =
56      LPhotEng / (TIMEST*boxes.boxLength*boxes.boxLength);;

57    bool firstCall = false;
58    bool lastCall = false;
```

```
59    string procId = "";
60  # if SIM_WITH_MPI
61    int id = comPtr->getId();
62    bool isTopCell = (id == comPtr->getNumProcessors()-1);
63    bool isBottomCell = (id == 0);
64    {
65      ostringstream sout;
66      sout << comPtr->getProcessorName() << ":"
67           << comPtr->getId() << "> " << flush;
68      procId = sout.str();
69    }
70  # endif

71    if ( VARIABLE_TIMESTEP ) {
72      // absorption probability
73      double tflt = 1. - Abs_prob / 100.0;
74      tflt        = pow (tflt, TIMEST);
75      abs_prob    = 1. - tflt;

76      LEngUnit    = Lpulsint * TIMEST * boxes.boxLength * boxes.boxLength ;
77    }

78    /* Gaussian time shape */
79    const double rel_time   = real_time - LAS_ON - LPULSLEN ;
80    const double gauss_exp  = rel_time * 2.0 / LPULSLEN;
81    const double puls_time  = exp(- gauss_exp * gauss_exp );


82    if ( LAS_ON-TIMEST <= real_time && real_time <= LAS_ON+TIMEST ) {
83      cout << SET_COLOR << procId
84           << "starting laser pulse (t=" << realtime << ")"
85           << RESET_COLOR << endl;
86      firstCall = true;
87      ++pulseCount;
88    }
89    if ( LAS_ON+2*LPULSLEN-TIMEST <= real_time &&
90         real_time <= LAS_ON+2*LPULSLEN+TIMEST ) {
91      cout << SET_COLOR << procId
92           << "stopping laser pulse (t=" << realtime << ")"
93           << RESET_COLOR << endl;
94      lastCall = true;
95    }

96    const double loc_energy_t    = puls_time * LEngUnit;
97    const double loc_intensity_t = puls_time * Lpulsint;

98    int not_absorbed = 0;
99    one_phot_abs = 0;
100   two_phot_abs = 0;
101   Numphot = 0;

102   int begin = 0;
```

```
103   int end = boxes.numBoxXY;

104   switch ( LSSHAPE ) {
105     case Laser::UNIFORM:
106       begin = 0;
107       end = boxes.numBoxXY;
108       break;
109     case Laser::GAUSSIAN:
110       begin = 1;
111       end = boxes.numBoxXY-1;
112       break;
113     default:
114       cerr << "(" << __FILE__ << ":" << __LINE__ << ") "
115            << "*** undefined laser spot shape ***" << endl
116            << " --> aborting..." << endl;
117       exit(1);
118   }

119   const int numSide = end-begin;

120   static Array2D<int> photonCount(numSide,numSide);
121 # if SIM_WITH_MPI
122   if ( !isTopCell) {
123     receiveFrom(photonCount, numSide, numSide, comPtr->getId()+1);

124     Array2D<int> tmp(3,1);
125     receiveFrom(tmp, 3, 1, comPtr->getId()+1);
126     one_phot_abs = tmp.at(0,0);
127     two_phot_abs = tmp.at(1,0);
128     Numphot = tmp.at(2,0);
129   }
130 # endif

131   const int ziMinBox = max(boxes.ziMin,boxes.firstNonDamped);
132   const int ziMaxBox = boxes.ziMax;
133 # if PRINT_ABS_PROFILE
134   vector<int> nPhotAbs(ziMaxBox-ziMinBox+1);
135   for ( unsigned int ii = 0; ii < nPhotAbs.size(); ++ii )
136     nPhotAbs[ii] = 0;
137 # endif


138   for (int cntx = begin; cntx < end; cntx ++)
139     for (int cnty = begin; cnty < end; cnty ++) {

140       double loc_energy_tl, loc_intensity_tl;
141       switch ( LSSHAPE ) {
142         case Laser::UNIFORM: {
143           loc_energy_tl = loc_energy_t;
144           loc_intensity_tl = loc_intensity_t;
145           break;
146         }
```

```
147           case Laser::GAUSSIAN: {
148             double distq;
149             double nStepX    = cntx ;
150             double nStepY    = cnty ;
151             nStepX += 0.5;
152             nStepY += 0.5;
153             distq  = (Center - nStepX * boxes.boxLength )
154               * (Center - nStepX * boxes.boxLength );
155             distq += (Center - nStepY * boxes.boxLength )
156               * (Center - nStepY * boxes.boxLength );
157             const double factor = exp (- distq / Lfocusradq );
158             loc_energy_tl = loc_energy_t * factor;
159             loc_intensity_tl = loc_intensity_t * factor;
160             break;
161           }
162           default:
163             cerr << "(" << __FILE__ << ":" << __LINE__ << ") "
164                  << "*** undefined laser spot shape ***" << endl
165                  << " --> aborting..." << endl;
166             exit(1);
167         };
168         totalLaserEnergy += loc_energy_tl;

169         int photons;
170 #       if SIM_WITH_MPI
171         if ( !isTopCell ) {
172           photons = photonCount.at(cntx,cnty);
173         }
174         else
175 #       endif
176         {
177           double tflt   = loc_energy_tl / LPhotEng;
178           photons   = (int) tflt;  // number of photons

179           { // correct for lost energy due to rounding
180             double tf1     = tflt - (double) photons;
181             double tf2     = tf1 * tf1;
182             if (myRandom.rand() < tf2 ) // check for two photon absorption
183               photons += 2;
184             else if (myRandom.rand() < tf1 )
185               photons += 1;
186           }

187           Numphot += photons ;
188         }

189         abs_probi    = Abs_probi * loc_intensity_tl ;
190         double abs_prob_ion = 1. - exp (abs_probi);

191         int remPhot   = photons; // remaining photons
192 #       if PRINT_ABS_PROFILE
193         int prevRemPhot = remPhot;
```

```
194 #     endif

195       for (int cntz = ziMaxBox; cntz >= ziMinBox; cntz --)
196       {
197         double loc_intensity = remPhot * intensity_unit;
198         abs_probi    = Abs_probi * loc_intensity;
199         abs_prob_ion = 1. - exp (abs_probi);

200         for (int cntw = 0; cntw < boxes.count[cntx][cnty][cntz]; cntw ++)
201         {
202           long ltint = boxes.box[cntx][cnty][cntz][cntw];
203           if (ltint > NAT)    /* check for electrons */
204             continue;


205           for (int cntp = 0; cntp < remPhot; cntp ++)
206           {
207             if ((cntp < remPhot - 1) && (myRandom.rand() < abs_prob_ion ))
208             {
209               state.exc[ltint]  += 2;
210               remPhot      -= 2;
211               two_phot_abs ++;
212             }
213             else if ((cntp < remPhot ) && (myRandom.rand() < abs_prob ))
214             {
215               ++state.exc[ltint];
216               --remPhot;
217               ++one_phot_abs;
218             }
219           }
220           if (state.exc[ltint] >= Ionnumphot )
221           {
222             state.exc[ltint] -= Ionnumphot;
223             ++state.ion[ltint];
224             ionize (ltint );
225           }
226         }
227       }
228       photonCount.at(cntx,cnty) = remPhot;
229       not_absorbed += remPhot;
230     }

231 # if SIM_WITH_MPI
232   if ( !isBottomCell) {
233     sendTo(photonCount, comPtr->getId()-1);

234     Array2D<int> tmp(3,1);
235     tmp.at(0,0) = one_phot_abs;
236     tmp.at(1,0) = two_phot_abs;
237     tmp.at(2,0) = Numphot;

238     sendTo(tmp, comPtr->getId()-1);
```

```
239   }
240 # endif


241 # if SIM_WITH_MPI
242   if ( isBottomCell )
243 # endif
244   {
245     if ( hfPtr != 0 )
246       hfPtr->laser(not_absorbed*intensity_unit/sqr(boxes.numBoxXY), TIMEST);

247     double energyJ = totalLaserEnergy * 1.6e-19;
248     double area = sqr(boxes.numBoxXY*boxes.boxLength * 1.0e-8); // cm2
249     double percentAbs = Numphot<=0 ? 0.0 :
250       ((double)(one_phot_abs+2*two_phot_abs))/((double)Numphot);
251     double energyAbs =
252       percentAbs * Numphot * (LPhotEng*PhysConv::eV_to_Joules) / area;
253     totalAbsorbedEnergy += energyAbs;
254   }
255 }
```