PARALLEL PROGRAMMING MODELS AND BOUNDARY INTEGRAL EQUATION METHODS FOR MICROSTRUCTURE ELECTROSTATICS

By

Frank Traenkle

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Science

(Chemical Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

1993

Abstract

The programming models presented by parallel computers are diverse and changing. We study the implementation of our application in different parallel programming models with a collaborative effort between chemical engineers and computer scientists.

The application considered is the class of three-dimensional elliptic partial differential equations (Laplace, Stokes, Navier) with solutions represented by boundary integral equations. These partial differential equations appear in basic microscopic descriptions of heterogenerous structured continua. As an example, we present results for the macroscopic dielectric constants and thermal conductivities of two-phase materials. The parallel algorithm follows naturally from our use of the Completed Double Layer Boundary Integral Equation Method (CDLBIEM).

The application is implemented in the message-passing programming model using the standard send-receive message-passing primitives in the CMMD library and the static shared-memory model in the form of Split-C, both running on the Thinking Machines CM-5 parallel computer. Furthermore, we study its implementation in a new parallel programming model — cooperative shared memory (CSM). Since CSM machines do not (yet) exist we evaluate our

application and machine designs with the Wisconsin Wind Tunnel (WWT), which runs CSM programs and calculates the performance of hypothetical parallel computers.

A major result is the demonstration that coding CDLBIEM is much simpler under CSM than with the message-passing model or Split-C, and yet performance (computational times and scaleup) is comparable, a fact that may be of great interest to designers of future machines.

Acknowledgements

I owe so many thanks to all my friends in Madison who supported me in writing this thesis and in enjoying Madison. Especially, I would like to thank Dannette M. Bousley, Gary A. Huber, Marc D. Jaros, Brian E. Saunders, and Steven S. William for proofreading this work.

I thank my parents, Fritz and Gerlinde Traenkle, and my brother, Goetz, for their love and financial support. Without their help, my stay in Madison would not have been possible.

I would like to thank all the members of my research group. Prof. Sangtae Kim has been giving me advice and support of any kind. My colleagues in alphabetical order, Douglas A. Brune, Linda A. Caine, Gary A. Huber, Dana J. Haselton, Iasonas G. Moustakis, Christine Maul, Peyman Pakdel, Saroja Ramanujan, Brian E. Saunders, and Zhengfang Xu, discussed with me all my questions about computers, mathematics, life and everything.

I also would like to thank my second advisor Prof. Mark D. Hill and several other people at the Computer Sciences Department, namely Babak Falsafi, Matthew I. Frank, Prof. James R. Larus, Alvin R. Lebeck, Shubhendu Mukherjee, Steven K. Reinhardt, Prof. Mary K. Vernon, and Prof. David A. Wood, for their advice concerning the Split-C and CSM codes and for

providing the Wisconsin Wind Tunnel.

Last but not least, I would like to thank Prof. M. Zeitz and the German Academic Exchange Service for arranging the exchange program between the Institut für Systemdynamik und Regelungstechnik at the University of Stuttgart – Germany and the Department of Chemical Engineering at the University of Wisconsin – Madison.

Nomenclature

Roman letters

$\boldsymbol{A}$, $\tilde{\boldsymbol{A}}$	system matrix
$\boldsymbol{C}_{kl}$	system submatrix
$\boldsymbol{D}$	electric displacement
E_P	efficiency
$\boldsymbol{E}^{\infty}$	electric field at infinity
$\boldsymbol{F}$	force
G	Green's function
$\bar{G}$	Green's function for container problem
K	double layer kernel
$\bar{K}$	double layer kernel for container problem
N	number of bodies
$N()$	nullspace
M	number of boundary elements
P	number of processors
Q	charge
Q_α	net charge of body α
S	boundary of the fluid or void domain
S_α	surface of body α
S_P	speedup
$\bar{S}_P$	modified speedup
$\boldsymbol{S}$	communication schedule (CS) matrix
V	interior domain (vacuum)
V_e	exterior domain (body)
V_∞	whole domain
a	sphere radius
c	volume fraction of inclusions
$\boldsymbol{b}$, $\tilde{\boldsymbol{b}}$	constant system vector
$\boldsymbol{d}_k$	constant system subvector
g_k	criterion for assigning body k to a processor
g^p	sum of criteria over all bodies on one processor

$\boldsymbol{n}$	surface normal pointing into the body
$\hat{\boldsymbol{n}}$	surface normal pointing from the body
$\boldsymbol{p}_k$	electric dipole moment of body k
s_{kl}	element of CS matrix
$\boldsymbol{x}$	position vector
$\boldsymbol{x}$	solution vector
$\boldsymbol{x}_\alpha$	position of singularity in body α
$\boldsymbol{y}_k$	solution subvector on body k

Greek letters

$\boldsymbol{\beta}$	electric susceptibility
δ	Dirac's delta function
δ_{ij}	Kronecker delta
$\boldsymbol{\epsilon}$	dielectric constant
ϵ_0	permittivity constant
$\boldsymbol{\eta}$	position vector on body surface
$\boldsymbol{\xi}$	position vector as integration variable
φ	double layer density
φ_{ki}	collocated double layer density on element i of body k
$\varphi^{(\alpha)}$	basis function of the nullspace on body α
ψ	electric potential
ψ_α	electric potential on body α
ψ^∞	electric potential at infinity

Mathematical operators

$\mathcal{H}$	Wielandt deflated double layer operator
$\mathcal{K}$	double layer operator
∇	gradient w.r.t. $\boldsymbol{x}$
$\nabla_{\boldsymbol{\xi}}$	gradient w.r.t. $\boldsymbol{\xi}$
$\nabla\cdot$	divergence

Subscripts

P	number of processors
i, j	boundary element label
k, l	body label
α	body label

Superscripts

p	processor label
(α)	body label
∞	domain at infinity
$*$	image

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computers and their programming models are developed by computer scientists. This thesis discusses an application which may influence this development. Application programmers and computer scientists have to cooperatively create parallel computers with one widely accepted programming model.

We solve N-body electrostatic interaction problems with boundary integral methods whose implementations exceed the limits of uniprocessor computers. In order to solve these problems, these methods are implemented in parallel MIMD (multiple-instruction and multiple-data) programming models on large-scale parallel machines. The features and performance of these implementations are studied.

1.1 Problem Definition

We solve elliptic PDEs for N-body problems. Examples for elliptic, time-independent PDEs are

- the Laplace equation

$$\nabla^2 \psi = 0 \tag{1.1}$$

  where ψ is the electric potential,

- the Stokes equations

$$-\nabla p + \mu \nabla^2 \boldsymbol{v} = \boldsymbol{0} \quad , \quad \nabla \cdot \boldsymbol{v} = 0 \tag{1.2}$$

  where $\boldsymbol{v}$ is the velocity and p the pressure, and

- the time-independent version of the Navier equation

$$(\lambda + \mu)\nabla(\nabla \cdot \boldsymbol{u}) + \mu \nabla^2 \boldsymbol{u} + \rho \boldsymbol{b} = \boldsymbol{0} \tag{1.3}$$

  where $\boldsymbol{u}$ is the elastic displacement [37].

This work discusses the numerical Completed Double Layer Boundary Integral Equation Method (CDLBIEM) which is presented for the Stokes equations in Kim & Karrila's Microhydrodynamics [31]. CDLBIEM can be applied to all three equations. For the first time, CDLBIEM is used to solve the Laplace equation.

The main purpose of this work is to optimize CDLBIEM on parallel machines and to investigate different programming models. In order to analyze the performance of the different implementations of CDLBIEM, the Laplace equation is chosen as the simplest case. Unlike the two other equations, the Laplace equation has a scalar variable as its unknown. By appropriate extensions of the optimized program for the Laplace equation, the computationally more

expensive Stokes and Navier equations can be solved.

The physical motivation for studying these equations is as follows. The Laplace equation describes the electric potential ψ in electrostatics [25] but is also the equation for the temperature T in time-independent heat conduction [6]. In the considered electrostatic problems, N charged bodies that are perfect conductors are distributed in an unbounded, 3-dimensional domain. CDLBIEM solves for the electric potential on the bodies' surfaces and in the surrounding medium. By using the method of images, we solve problems where the bodies are inclusions in a macroscopic medium which is surrounded by a spherical perfect conductor. The solution returned by CDLBIEM is used to calculate the macroscopic dielectric constant of this two-phase material [26, 53]. Because of the analogy of the electrostatics to time-independent heat conduction, this dielectic constant is equal to the thermal conductivity of a two-phase material, where the N bodies are perfectly conducting inclusions in a medium with a finite thermal conductivity surrounded by a perfectly conducting container [4].

The Stokes equations describe the creeping flow of Newtonian fluids with very high viscosity or negligible inertia [31]. CDLBIEM solves the mobility problem where the forces and torques acting on the particles are given and the velocity and the rigid body motion of the particles are to be calculated.

A simultaneous solution of the Stokes and the Laplace equations is of great importance in the form of electrorheological fluid simulations. An electrorheological fluid is a suspension of polarizable particles in a nonconducting oil [32]. Its outstanding rheological property is the change of its viscosity, which is of orders of magnitude greater under an applied electric

field. This feature makes electrorheological fluids very interesting for applications in mechanical systems such as clutches, engine mounts, shock absorbers, hydraulic valves, among others.

1.2 Parallel Computing

CDLBIEM is implemented on three programming models: the message-passing model, the static shared-memory model with non-uniform memory access (NUMA) in the form of Split-C, and the cooperative shared memory model (CSM). The first two models run on a Thinking Machines CM-5, the third model runs on a cache-coherent distributed memory machine with the cache-coherence protocol Dir_1SW. However, since this computer does not (yet) exist, it is simulated as a virtual prototype on the Wisconsin Wind Tunnel (WWT) which also runs on a Thinking Machines CM-5.

1.3 Summary of Results

CDLBIEM is a fast numerical method with a high numerical accuracy. In the case of two almost touching spheres, its relative error is less than half a percent. Although our programs can solve for the double layer density on bodies with arbitrary shapes, we use spheres as the simplest case. By implementing a scheme which controls the communication between the processors, the iteration time is reduced by a factor of around three for problem sizes examined in this study. This communication scheme corresponds to the physical interactions between the bodies and reduces the amount of communication and computation. Especially the reduced amount of

communication is of great advantage for implementations on future parallel machines, which will have a even greater discrepancy between their computation and communication performance.

The parallel implementations of CDLBIEM have performed well. However, scaling results are available only for the implementation on CSM. The speedup of the program is 7.7 from running it on 16 processors to running it on 128 processors.

By using the method of images, we can solve for the dielectric constants and thermal conductivities of two-phase materials. The computational predictions of these properties match closely the analytical solutions obtained by Zuzovsky and Brenner [53] (error of less than 0.6% for small volume fractions c). The amount of computation is increased by one order of magnitude, but the amount of communication and the memory usage stays constant because the number of unknowns does not increase.

1.4 Organization

Chapter 2 gives an overview of the electrostatics according to Jackson's book [25], and the Poisson equation and its special form, the Laplace equation, are derived.

In Chapter 3 the Completed Double Layer Integral Equation Method for the Laplace equation is presented. The Laplace equation is transformed to its classical boundary integral representation which includes a single layer and a double layer integral. This equation is modified to the completed double layer boundary integral equation wherein the double layer operator is made invertible by completing its range. In addition, Wielandt's deflation reduces its spectral radius from 1 to 1/3 in the case of one unit sphere. The resulting boundary integral equation

for the unknown double layer density is a Fredholm integral equation of the second kind.

In Chapter 4 we discretize the boundaries of the bodies into boundary elements, collocate the double layer density, and approximate the integrals by Gauss-Legendre quadrature. This yields a linear algebraic equation system for the unknown double layer density.

In Chapter 5 we first discuss different aspects of parallel computers, such as their architecture and their programming models. Then we give an introduction to the additional criteria of parallel programming. In order to solve the linear algebraic equation system, we use Jacobi iteration. This algorithm is parallelized and, by using a communication schedule, extended to an asynchronous iteration with accelerated convergence properties similar to the sequential Gauss-Seidel iteration. Afterwards, the load balance and the implementation of this algorithm on the three different programming models are discussed.

In order to solve for physical properties of two-phase materials, we use the method of images in Chapter 6 which yields a modified boundary integral equation for N bodies that are inclusions in a spherical perfect conductor. Our program solves this boundary integral equation and returns the double layer density on the boundaries of the bodies, which in turn can be manipulated to yield the macroscopic dielectric constant and the macroscopic thermal conductivity of the two-phase medium.

In Chapter 7 we analyze the performance of the implementations on the three different programming models. Furthermore, we present results for the electric potential and for the physical properties of two-phase materials.

The final chapter, Chapter 8, summarizes this thesis and proposes future work.

Chapter 2

Classical Electrostatics

Classical electrostatics provides the physical foundation for the problems considered in this work. Following the book by J.D. Jackson [25], the fundamental equations of electrostatics are presented and *Poisson's equation* and its special form *Laplace's equation* are derived which describe the electric potential in a vacuum with or without charges.

Since the systems considered in our problems are assumed to be in equilibrium or quasi-static, there is no time-dependence and electrostatics can be applied instead of electrodynamics. Electrostatics deals with macroscopic phenomena. Thus, point charges or electric fields at a point must be viewed as mathematical constructs that permit a description of the phenomena at the macroscopic level, but may fail to have meaning microscopically.

The basic two equations from which all other equations in this chapter can be derived, are *Coulomb's law* and *Gauss' law*. The next section introduces Coulomb's law and defines both the electric field and the electric potential.

2.1 Coulomb's Law

Coulomb's law describes the force between two charged bodies at rest with respect to each other. Coulomb discovered experimentally that the force between two charged bodies separated in air by a large distance compared to their dimensions

- varied proportionally to the magnitude of each charge,
- varied inverse proportionally to the square of the distance between them,
- is directed along the straight line joining the charges,
- is attractive, if the bodies are oppositely charged; and repulsive, if the bodies have the same type of charge.

Based on his observations the following law for the force $\boldsymbol{F}$ between two charges in a vacuum is stated:

$$\boldsymbol{F} = \frac{1}{4\pi\epsilon_0}\frac{q_1 q_2\left(\boldsymbol{x}_1 - \boldsymbol{x}_2\right)}{\left|\boldsymbol{x}_1 - \boldsymbol{x}_2\right|^3} \tag{2.1}$$

where

- q_1, q_2 are the negative or positive magnitudes of the point charges,
- $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ are the 3-dimensional positions of the charges,
- $\epsilon_0 = \frac{10^7}{4\pi c^2}\frac{mF}{s^2}$ is the permittivity constant (vacuum) where c is the speed of light[1]. For the same charges interacting in a dielectric medium, we usually observe a smaller force, and

[1]The equations given in this chapter are in the MKSA unit system.

so Coulomb's law is modified by increasing the permittivity with a scale factor known as the *dielectric constant* ϵ.

It is seen that the interactions between one charge and other different charges are superimposed linearly. Although the interaction is measured by a force, a useful concept is to introduce a potential field, called the *electric field*. A test charge q at a given point position in an electic field $\boldsymbol{E}$ experiences the force

$$\boldsymbol{F} = \boldsymbol{E}q \ . \tag{2.2}$$

This is the definition of the electric field $\boldsymbol{E}$. The test charge must be negligibly small, so that it does not disturb the electric field. Combining the equations (2.1) and (2.2) the electric field at a point $\boldsymbol{x}$ due to a point charge q_1 at the position $\boldsymbol{x}_1$ is

$$\boldsymbol{E}(\boldsymbol{x}) = \frac{1}{4\pi\epsilon_0} \frac{q_1 \left(\boldsymbol{x} - \boldsymbol{x}_1\right)}{\left|\boldsymbol{x} - \boldsymbol{x}_1\right|^3} \ . \tag{2.3}$$

The experimentally observed linear superposition of forces due to many point charges q_i at the positions $\boldsymbol{x}_i$ results in the electric field

$$\boldsymbol{E}(\boldsymbol{x}) = \frac{1}{4\pi\epsilon_0} \sum_{i=1}^{N} \frac{q_i \left(\boldsymbol{x} - \boldsymbol{x}_i\right)}{\left|\boldsymbol{x} - \boldsymbol{x}_i\right|^3} \ . \tag{2.4}$$

The *generalized Coulomb's law* is obtained introducing the *charge density* at a point $\boldsymbol{x}$

$$\rho(\boldsymbol{x}) = \lim_{\Delta V \to 0} \frac{\Delta q(\boldsymbol{x})}{\Delta V} \ , \tag{2.5}$$

which is measured in $\frac{C}{m^3}$ using MKSA units. Replacing the sum in Equation (2.4) by a volume integral yields the generalized Coulomb's law:

$$\boldsymbol{E}(\boldsymbol{x}) = \frac{1}{4\pi\epsilon_0}\int_V \frac{\rho(\boldsymbol{\xi})\,(\boldsymbol{x}-\boldsymbol{\xi})}{|\boldsymbol{x}-\boldsymbol{\xi}|^3}\,dV_{\boldsymbol{\xi}}\ . \tag{2.6}$$

Note: this is the electric field created only by a distribution of point charges. The additional electric field induced by electric dipoles is discussed in Chapter 3.

The electric field is irrotational. This fact may be derived from Equation (2.6). The vector factor in the integrand is the negative gradient of the scalar $1/|\boldsymbol{x}-\boldsymbol{\xi}|$ w.r.t. $\boldsymbol{x}$:

$$\frac{\boldsymbol{x}-\boldsymbol{\xi}}{|\boldsymbol{x}-\boldsymbol{\xi}|^3} = -\nabla\frac{1}{|\boldsymbol{x}-\boldsymbol{\xi}|}\ . \tag{2.7}$$

Thus, Equation (2.6) becomes

$$\boldsymbol{E}(\boldsymbol{x}) = -\frac{1}{4\pi\epsilon_0}\nabla\int \frac{\rho(\boldsymbol{\xi})}{|\boldsymbol{x}-\boldsymbol{\xi}|}\,d(V_{\boldsymbol{\xi}})\ . \tag{2.8}$$

But the curl of every gradient is equal to zero, so that

$$\nabla\times\boldsymbol{E} = \boldsymbol{0}\ . \tag{2.9}$$

As shown in Equation (2.8) the electric field can be written as the gradient of a scalar. This scalar function is named the *electric potential* $\psi(\boldsymbol{x})$. The electric field is defined as the negative

negative of this gradient:

$$\boldsymbol{E}(\boldsymbol{x}) = -\nabla\psi(\boldsymbol{x}) \ . \tag{2.10}$$

When a test charge is moved in an electric field, work is done. If the test charge is moved from a point $\boldsymbol{x}_0$ to a point $\boldsymbol{x}$ along a curve C, then this work is given by the following line integral:

$$W = -\int_C \boldsymbol{F} \cdot d\boldsymbol{s} = -q\int_C \boldsymbol{E} \cdot d\boldsymbol{s} \ . \tag{2.11}$$

The minus sign indicates that the work is done against the action of the field. Replacing $\boldsymbol{E}$ by the negative gradient of the potential yields that the integral is path independent:

$$W = q\int_C \nabla\psi \cdot d\boldsymbol{s} = q\int_C d\psi = q\int_{\boldsymbol{x}_0}^{\boldsymbol{x}} d\psi = q\left(\psi\,|_{\boldsymbol{x}} - \psi\,|_{\boldsymbol{x}_0}\right) \ . \tag{2.12}$$

2.2 Gauss' Law

In this section the second fundamental equation of electrostatics is presented: *Gauss' law*. To specify a potential field completely, the divergence and the curl of the field must be given. So far only the curl is given in Equation (2.9) as derived from Coulomb's law. In addition, Gauss' law specifies the divergence of the field.

Consider a point charge with magnitude q placed inside a closed surface S. Gauss' law states that the integral over this surface, with the normal vector $\boldsymbol{n}$ pointing outside, evaluates

to:

$$\oint_S \boldsymbol{E} \cdot \boldsymbol{n}\, dS = \begin{cases} \frac{q}{\epsilon_0} & \text{if } q \text{ inside } S \\ 0 & \text{if } q \text{ outside } S\ . \end{cases} \tag{2.13}$$

If there is a continuous *charge density* $\rho(\boldsymbol{x})$ inside the closed surface S, Gauss' law becomes:

$$\oint_S \boldsymbol{E} \cdot \boldsymbol{n}\, dS = \frac{1}{\epsilon_0} \int_V \rho(\boldsymbol{x})\, dV \tag{2.14}$$

Gauss' law may be written in differential form applying the *divergence theorem*:

$$\oint_S \boldsymbol{E} \cdot \boldsymbol{n}\, dS = \int_V \nabla \cdot \boldsymbol{E}\, dV\ . \tag{2.15}$$

Since this equation is valid for any arbitrary volume V, both integrands must be equal:

$$\nabla \cdot \boldsymbol{E} = \frac{\rho}{\epsilon_0}\ . \tag{2.16}$$

This is Gauss' law in differential form.

Using the fact that $\boldsymbol{E} = -\nabla \cdot \psi$, we obtain *Poisson's equation*:

$$\nabla^2 \psi = -\frac{\rho}{\epsilon_0}\ . \tag{2.17}$$

If there is no charge, Poisson's equation reduces to *Laplace's equation*:

$$\nabla^2 \psi = 0\ . \tag{2.18}$$

From now on we use *electrostatic units* instead of MKSA units[2]. Thus, Poisson's equation becomes:

$$\nabla^2\psi = -4\pi\rho(\boldsymbol{x}) \ . \tag{2.19}$$

[2]For a brief discussion of units see Appendix A.

Chapter 3

Boundary Integral Equation Methods for Laplace's Equation

In this chapter boundary integral equations (BIEs) are derived that model the electric potential generated by N fixed charged bodies[1] with arbitrary shapes. First of all these bodies are located in an unbounded infinite void domain (in Chapter 6, we consider N bodies placed inside a spherical perfect conductor).

Laplace's equation derived in the previous chapter is transformed by an integral transformation. The resulting equations are boundary integral equations that are equivalent to the differential Laplace equation but have the advantage that their unknowns are densities confined to the body surfaces, i.e., in going from the PDE to the BIE there is a reduction in dimensionality. In addition, the *completed double layer-boundary integral equation method (CDLBIEM)*

[1]The terms *body*, *particle* and *charge* are used interchangeably throughout this work.

leads to an integral equation that can be discretized into well-posed linear algebraic equations (see Chapter 4), and is amenable to solution by fixed-point iterative methods (see Chapter 5).

First, the solution of Poisson's equation for a point charge in a vacuum is presented, which yields the Green's function. Second, CDLBIEM is presented in detail for the single-body geometry. Finally, the equations are extended to the multi-body geometry in a straightforward manner.

The underlying assumption is that the medium surrounding the bodies is a vacuum, so that the dielectric constant of the medium is equal to 1.[2] Furthermore all bodies are perfect conductors, i.e., their surfaces are equipotentials.

3.1 Integral Transformation

In this section we present Green's function of the Poisson and Laplace equations. First, this Green's function is shown to be the solution of Poisson's equation in case of a point charge. Second, it is used as the kernel of the integral transformation to transform Laplace's equation into a boundary integral equation. The analysis is presented for one arbitrarily shaped body and yields the classical integral representation for Laplace's equation that can be used directly as a computational method. However, in the subsequent sections a far more efficient computational method based on the double layer potential is derived. Finally, in Section 3.5 the analysis is extended to N bodies.

[2] The analysis is readily extended to a dielectric medium by introducing a dielectric constant not-equal to 1.

3.1.1 Green's Function

As a small example for the application of the integral transformation, consider a point charge of magnitude Q placed at the origin in an unbounded infinite space V_∞ and a superimposed time-independent ambient electric field, which causes the ambient potential $\psi^\infty(\boldsymbol{x})$. The charge density is given by Dirac's delta function scaled with Q:

$$\rho(\boldsymbol{x}) = Q\delta(\boldsymbol{x}) \quad ; \boldsymbol{x} \in V_\infty \ . \tag{3.1}$$

Inserting this charge density in Poisson's equation (2.17), the governing equations for the electric potential created by one point charge Q placed at the origin are obtained:

$$\begin{aligned} &\text{PDE} && \nabla^2\psi = -4\pi Q\delta(\boldsymbol{x}) && \boldsymbol{x} \in V_\infty \\ &\text{BC} && \psi\,\big|_{|\boldsymbol{x}|\to\infty} = \psi^\infty(\boldsymbol{x}) \\ &\text{integrability} && \nabla \times \nabla\psi = \boldsymbol{0} && \boldsymbol{x} \in V_\infty \ . \end{aligned} \tag{3.2}$$

$\psi^\infty(\boldsymbol{x})$ is an ambient potential field, e.g. for a constant ambient electric field $\boldsymbol{E}^\infty$ this potential is given by $\psi^\infty(\boldsymbol{x}) = -\boldsymbol{E}^\infty \cdot \boldsymbol{x}$.

Green's function of this boundary value problem is given by:

$$G(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi\,|\boldsymbol{\xi} - \boldsymbol{x}|} \tag{3.3}$$

and $G(\boldsymbol{x}, \boldsymbol{\xi})$ satisfies the following homogeneous equations:

$$\begin{aligned} &\text{PDE} && \nabla_{\xi}^2 G = -\delta(\boldsymbol{\xi} - \boldsymbol{x}) \quad \boldsymbol{\xi} \in V_\infty \\ &\text{BC} && G\,\big|_{|\boldsymbol{\xi}|\to\infty} = 0 \end{aligned} \tag{3.4}$$

$$\text{integrability} \quad \nabla_{\boldsymbol{\xi}} \times \nabla_{\boldsymbol{\xi}} G = \mathbf{0} \qquad \boldsymbol{\xi} \in V_\infty \ .$$

The Equations (3.2) are transformed by the following integral transformation with $G(\boldsymbol{x}, \boldsymbol{\xi})$ as its kernel:

$$\int_{V_\infty} G(\boldsymbol{x}, \boldsymbol{\xi}) \bullet \, dV_{\boldsymbol{\xi}} \ . \tag{3.5}$$

The transformed equation is:

$$\int_V G(\boldsymbol{x}, \boldsymbol{\xi}) \nabla_{\boldsymbol{\xi}}^2 \psi(\boldsymbol{\xi}) \, dV_{\boldsymbol{\xi}} = -4\pi Q G(\boldsymbol{x}, \mathbf{0}) \ . \tag{3.6}$$

To use the boundary conditions the volume integral on the left-hand side is evaluated by *Green's theorem*, also known as *Green's second identity*[3]:

$$\int_V \left(\nabla_{\boldsymbol{\xi}}^2 G\psi(\boldsymbol{\xi}) - G\nabla_{\boldsymbol{\xi}}^2 \psi(\boldsymbol{\xi}) \right) \, dV_{\boldsymbol{\xi}} = \oint_{S_\infty} (\nabla_{\boldsymbol{\xi}} G\psi - G\nabla_{\boldsymbol{\xi}}\psi) \cdot \boldsymbol{n}(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \tag{3.7}$$

where $S_\infty = \partial V_\infty$ is the boundary of the unbounded space at infinity. Inserting the boundary

[3] Green's theorem corresponds to *intergration by parts* in one dimension and *Lorentz's reciprocal theorem* for vector fields.

conditions of Equation (3.2) into the boundary integral over S yields the solution for ψ:

$$\begin{aligned}\psi(\boldsymbol{x}) &= 4\pi Q G(\boldsymbol{x},\mathbf{0}) + \underbrace{\oint_{S_\infty} (G\nabla_{\boldsymbol{\xi}}\psi - \psi\nabla_{\boldsymbol{\xi}}G)\cdot\boldsymbol{n}(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}}}_{=\psi^\infty(\boldsymbol{x})} \\ &= \psi^\infty(\boldsymbol{x}) + \frac{Q}{|\boldsymbol{x}|}\ . \end{aligned} \tag{3.8}$$

Note that the Green's function $G(\boldsymbol{x},\mathbf{0})$ is the fundamental solution of both the Poisson and Laplace equations.

3.1.2 Charge with Arbitrary Surface

Now we consider one arbitrarily shaped body with a closed surface $S = \partial V$. The whole domain is denoted by V_∞. The vacuum enclosing the charge is denoted by V, further on called the *interior*. Therefore the domain of the body is $V_e = V_\infty - V$, further on called the *exterior*. Since there is no other charge in the vacuum, the governing equations in V are:

$$\begin{array}{lll} \text{PDE} & \nabla^2\psi = 0 & \boldsymbol{x}\in V^0 \\ \text{BC} & \psi\,\big|_{|\boldsymbol{x}|\to\infty} = \psi^\infty(\boldsymbol{x}) & \\ & & \\ \text{integrability} & \nabla\times\nabla\psi = \mathbf{0} & \boldsymbol{x}\in V^0\ . \end{array} \tag{3.9}$$

The Green's function for this problem is the same as in the previous section. Transforming these equations with the integral transformation yields the following integral equation:

$$\int_V G(\boldsymbol{x},\boldsymbol{\xi})\nabla_{\boldsymbol{\xi}}^2\psi(\boldsymbol{\xi})\,dV_{\boldsymbol{\xi}} = 0\ . \tag{3.10}$$

Again we use *Green's theorem* and get:

$$\int_V \nabla_{\boldsymbol{\xi}}^2 G\psi(\boldsymbol{\xi})\,dV_{\boldsymbol{\xi}} + \oint_{S+S_\infty} (G\nabla_{\boldsymbol{\xi}}\psi - \nabla_{\boldsymbol{\xi}}G\psi)\cdot\boldsymbol{n}(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} = 0 \tag{3.11}$$

where $\boldsymbol{n}$ is the surface normal on the particle pointing out of the integration domain into the particle. Evaluating the first integral yields the potential on the body surface and in the integration domain:

$$\int_V \underbrace{\nabla_{\boldsymbol{\xi}}^2 G}_{=-\delta(\boldsymbol{\xi}-\boldsymbol{x})}\ \psi(\boldsymbol{\xi})\,dV_{\boldsymbol{\xi}} = \begin{cases} -\psi(\boldsymbol{x}) & \text{if } \boldsymbol{x}\in V^0 \\ 0 & \text{if } \boldsymbol{x}\notin \overline{V} \\ -\frac{1}{2}\psi(\boldsymbol{x}) & \text{if } \boldsymbol{x}\in S\ . \end{cases} \tag{3.12}$$

As $\boldsymbol{x}$ approaches the surface S from both sides, the surface looks locally planar. The jump in the potential is equal in magnitude whether $\boldsymbol{x}$ approaches from the interior or from the exterior because of symmetry. Therefore the result on the surface is $-\frac{1}{2}\psi(\boldsymbol{x})$.

The second integral is an integral both over the body surface and the space boundaries at

infinity. By applying the boundary conditions both for ψ and for G we get:

$$\oint_{S_\infty} (G\nabla_{\boldsymbol{\xi}}\psi - \nabla_{\boldsymbol{\xi}}G\psi)\cdot \boldsymbol{n}(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = \psi^{\infty}(\boldsymbol{x}) \ . \tag{3.13}$$

By inserting the equations (3.12) and (3.13) into Green's theorem (3.11) yields the final solution of this section:

$$\left.\begin{array}{ll} \psi(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in V^0 \\ 0 & \text{if } \boldsymbol{x} \notin \overline{V} \\ \frac{1}{2}\psi(\boldsymbol{x}) & \text{if } \boldsymbol{x} \in S \end{array}\right\} = \psi^{\infty}(\boldsymbol{x}) - \underbrace{\oint_S G(\boldsymbol{x},\boldsymbol{\xi})\hat{\boldsymbol{n}}(\boldsymbol{\xi})\cdot\nabla_{\boldsymbol{\xi}}\psi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}}}_{\text{single layer integral}}$$
$$+ \underbrace{\oint_S \nabla_{\boldsymbol{\xi}}G(\boldsymbol{x},\boldsymbol{\xi})\cdot\hat{\boldsymbol{n}}(\boldsymbol{\xi})\psi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}}}_{\text{double layer integral}} \tag{3.14}$$

where $\hat{\boldsymbol{n}} = -\boldsymbol{n}$ is the normal vector on the surface pointing out of the conductor and into the vacuum.

Inside the conductor the potential is zero, that is, there is no electric field, since the interior of a charged surface is a Faraday's cage, as long it is a perfect conductor.

There are three contributions to the potential as shown in Equation (3.14):

- The ambient potential $\psi^{\infty}(\boldsymbol{x})$ can be invoked by a constant electric field, $\boldsymbol{E}^{\infty}$ for instance, and thus $\psi^{\infty}(\boldsymbol{x}) = -\boldsymbol{E}^{\infty}\cdot\boldsymbol{x}$.

- The first integral is the *single layer potential*, that corresponds physically to a surface distribution of charges.

- The second integral, the *double layer potential*, is a potential created by a surface distribution of electric dipoles.

Using Equation (3.14) as a computational method is not recommended, since its discretization leads to linear algebraic equations with a dense system matrix. This system of equations has to be solved by algorithms such as Gaussian elemination, which are notoriously difficult and inefficient to implement on parallel computers.

Furthermore (3.14) is a *Fredholm equation of the first kind* [15, 45], that is the unknown ψ only appears under the integrals. In the sense of Hadamard this equation is *ill-posed*, which can result in convergence problems especially when fine meshes for discretization are used. The ill-posedness is caused by the compactness of the single-layer operator and its unbounded inverse, so that small changes in the input are mapped by the unbounded inverse operator to large changes in the output. Therefore an alternate integral representation is approached, which yields a well-posed *Fredholm equation of the second kind* and a highly-parallel algorithm.

3.2 Properties of the Double Layer Operator

Before we derive the completed integral representation, we show some properties of the *double layer operator*, which describes the influence of a surface distribution of electric dipoles on the electric potential, as in Equation (3.14). There is still just one particle considered, whereas in Section 3.5 the equations for N particles are derived.

The double layer operator is basically the double layer potential of Equation (3.14):

$$\mathcal{K}\psi(\boldsymbol{x}) := 2 \oint_S \nabla_{\boldsymbol{\xi}} G(\boldsymbol{x}, \boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi}) \psi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} = \oint_S K(\boldsymbol{x}, \boldsymbol{\xi}) \psi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \tag{3.15}$$

where $\boldsymbol{x} \in S$. The *kernel* of the double layer operator is defined as:

$$K(\boldsymbol{x}, \boldsymbol{\xi}) = 2\nabla_{\boldsymbol{\xi}} G(\boldsymbol{x}, \boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi}) = \frac{\hat{\boldsymbol{n}}(\boldsymbol{\xi}) \cdot (\boldsymbol{x} - \boldsymbol{\xi})}{2\pi \left| \boldsymbol{x} - \boldsymbol{\xi} \right|^3} \, . \tag{3.16}$$

Note that $\mathcal{K}\psi(\boldsymbol{x})$ is scaled by 2, so that the subsequent equations may become more concise.

3.2.1 Jump Properties

The jump properties of the double layer potential are important and are derived directly from the integral representation of $\psi(\boldsymbol{x})$, Equation (3.14). The single layer potential and the ambient potential are continuous across the particle surface S, thus the jump in the potential is caused by the double layer potential alone. Let $\boldsymbol{x} = \boldsymbol{\eta} + \epsilon \hat{\boldsymbol{n}}$ with $\boldsymbol{\eta} \in S$ and $|\epsilon| \ll 1$, so that

$$\begin{aligned} \boldsymbol{x} \in V^0 \quad &\text{corresponds to} \quad \epsilon > 0 \, , \\ \boldsymbol{x} \in S \quad &\text{corresponds to} \quad \epsilon = 0 \, , \text{and} \\ \boldsymbol{x} \notin \overline{V} \quad &\text{corresponds to} \quad \epsilon < 0 \, . \end{aligned} \tag{3.17}$$

Inserting this substitution in Equation (3.14) yields the *jump properties* of the double layer operator:

$$\lim_{\epsilon \to 0+} \oint_S K(\boldsymbol{x}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} + \varphi(\boldsymbol{\eta}) \tag{3.18}$$

$$\lim_{\epsilon \to 0-} \oint_S K(\boldsymbol{x}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} - \varphi(\boldsymbol{\eta})\ . \tag{3.19}$$

3.2.2 Adjoint Operator

The *adjoint operator* $\mathcal{K}^*$ of the linear operator $\mathcal{K}$ is defined as follows:

$$\langle \mathcal{K}\varphi_1, \varphi_2 \rangle = \langle \varphi_1, \mathcal{K}^* \varphi_2 \rangle \tag{3.20}$$

where $\langle f, g \rangle$ denotes the *inner product*

$$\langle f, g \rangle = \oint_S f g\, dS \tag{3.21}$$

of the functions f and g defined on the surface S. Thus the transformation with the operator $\mathcal{K}$ inside an inner product may be flipped to the other side of the inner product by changing the operator to $\mathcal{K}^*$.

We are going to prove that the normal derivative of the single layer operator is the adjoint operator of the double layer operator. The potential $\psi(\boldsymbol{x})$ is given by the single layer operator

applied to the charge density $\sigma(\boldsymbol{x})$ on the particle surface S:

$$\psi(\boldsymbol{x}) = \frac{1}{4\pi} \oint_S \frac{\sigma(\boldsymbol{\xi})}{|\boldsymbol{x} - \boldsymbol{\xi}|} \, dS_{\boldsymbol{\xi}} \; . \tag{3.22}$$

The normal derivative of $\psi(\boldsymbol{x})$ on the particle surface is given by:

$$\begin{aligned} \frac{\partial \psi(\boldsymbol{x})}{\partial \hat{\boldsymbol{n}}} = \hat{\boldsymbol{n}}(\boldsymbol{x}) \cdot \nabla \psi(\boldsymbol{x}) &= \frac{1}{4\pi} \oint_S \frac{\hat{\boldsymbol{n}}(\boldsymbol{x}) \cdot (\boldsymbol{\xi} - \boldsymbol{x})}{|\boldsymbol{x} - \boldsymbol{\xi}|^3} \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \\ &= \frac{1}{2} \oint_S K(\boldsymbol{\xi}, \boldsymbol{x}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \\ &= \frac{1}{2} \oint_S K^*(\boldsymbol{x}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \; . \end{aligned} \tag{3.23}$$

Thus the adjoint kernel is given by $K^*(\boldsymbol{x}, \boldsymbol{\xi}) = K(\boldsymbol{\xi}, \boldsymbol{x})$, only the two independent variables $\boldsymbol{x}$ and $\boldsymbol{\xi}$ have to be swapped. Note that since $K^*(\boldsymbol{x}, \boldsymbol{\xi}) \neq K(\boldsymbol{x}, \boldsymbol{\xi})$, the double layer operator $\mathcal{K}$ is not self-adjoint.

Like the double layer operator $\mathcal{K}$, its adjoint operator $\mathcal{K}^*$ is discontinous crossing the particle surface S. However, the sum $\mathcal{K}(\sigma) + \mathcal{K}^*(\sigma)$ is a continuous function when crossing S, because the combined kernel is no longer weakly singular. Thus, according to the jump properties of $\mathcal{K}$ in Section 3.2.1 the *jump properties* of $\mathcal{K}^*$ are

$$\lim_{\epsilon \to 0+} \oint_S K^*(\boldsymbol{x}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} = \oint_S K^*(\boldsymbol{\eta}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} - \sigma(\boldsymbol{\eta}) \tag{3.24}$$

$$\lim_{\epsilon \to 0-} \oint_S K^*(\boldsymbol{x}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} = \oint_S K^*(\boldsymbol{\eta}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} + \sigma(\boldsymbol{\eta}) \tag{3.25}$$

with $\boldsymbol{x} = \boldsymbol{\eta} + \epsilon \hat{\boldsymbol{n}}$ and $\boldsymbol{\eta} \in S$.

3.2.3 Eigenvalues for The Unit Sphere

In this section the eigenvalues of the double layer operator for one unit sphere are derived. Since the double layer operator is compact it has a discrete spectrum of eigenvalues, with at most one accumulation point (see Theorem 5.3.2 in [15]). Let $\varphi(\boldsymbol{x})$ be an eigenfunction of the double layer operator $\mathcal{K}$ and λ its corresponding eigenvalue, that is:

$$\mathcal{K}\varphi(\boldsymbol{x}) = \oint_S K(\boldsymbol{x},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = \lambda\varphi(\boldsymbol{x}) \ . \tag{3.26}$$

There is a system of 2 *characteristic equations*. The first equation is obtained by imposing the condition that the electric field is continuous across surface S:

$$\boldsymbol{E}^{(e)}(\boldsymbol{\eta}) := \lim_{\epsilon\to 0-} \boldsymbol{E}(\boldsymbol{x}) = \lim_{\epsilon\to 0+} \boldsymbol{E}(\boldsymbol{x}) =: \boldsymbol{E}^{(i)}(\boldsymbol{\eta}) \tag{3.27}$$

where $\boldsymbol{x} = \boldsymbol{\eta} + \epsilon\hat{\boldsymbol{n}}$ and $\boldsymbol{E}^{(i)}$ and $\boldsymbol{E}^{(e)}$ are the interior ($\boldsymbol{x} \in V^0$) and exterior ($\boldsymbol{x} \notin \overline{V}$ or $\boldsymbol{x} \in V_e$) electric fields. The solution of Laplace's equation for one charged unit sphere may be written in spherical harmonics [25]:

$$\psi_n^{(e)}(r,\theta,\phi) = a_n r^n \sum_{m=-n}^{n} P_n^m(\cos\theta)e^{im\phi} \tag{3.28}$$

$$\psi_n^{(i)}(r,\theta,\phi) = A_n r^{-n-1} \sum_{m=-n}^{n} P_n^m(\cos\theta)e^{im\phi} \tag{3.29}$$

where $\psi_n^{(i)}$ is one mode of the potential in the interior region ($\boldsymbol{x} \in V^0$) and $\psi_n^{(e)}$ one mode in the exterior region ($\boldsymbol{x} \notin \overline{V}$). Here P_n^m are the *Associated Legendre Functions*. Since $\boldsymbol{E} = -\nabla\psi$,

(3.27) can be written in terms of $\psi_n^{(e)}$ and $\psi_n^{(i)}$:

$$\left.\frac{\partial \psi_n^{(e)}}{\partial r}\right|_{r=1} = \left.\frac{\partial \psi_n^{(i)}}{\partial r}\right|_{r=1} \quad . \tag{3.30}$$

With the harmonic solutions (3.28) and (3.29) this leads to the first characteristic equation:

$$\kappa_n := \frac{\psi_n^{(e)}(\boldsymbol{\eta})}{\psi_n^{(i)}(\boldsymbol{\eta})} = -\frac{n+1}{n} \quad . \tag{3.31}$$

The second characteristic equation is obtained from the jump conditions (3.18) and (3.19). Let $\psi^{(i)}$ and $\psi^{(e)}$ denote the interior and exterior double layer potential. Note: the single layer potential and the ambient potential are not taken into account here. From the jump conditions we get:

$$\psi_n^{(i)}(\boldsymbol{\eta}) = \frac{1}{2}\oint_S K(\boldsymbol{\eta},\boldsymbol{\xi})\varphi_n(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} + \frac{1}{2}\varphi_n(\boldsymbol{\eta}) = \frac{1}{2}(\lambda_n+1)\varphi_n(\boldsymbol{\eta}) \tag{3.32}$$

$$\psi_n^{(e)}(\boldsymbol{\eta}) = \frac{1}{2}\oint_S K(\boldsymbol{\eta},\boldsymbol{\xi})\varphi_n(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} - \frac{1}{2}\varphi_n(\boldsymbol{\eta}) = \frac{1}{2}(\lambda_n-1)\varphi_n(\boldsymbol{\eta}) \ , \tag{3.33}$$

since $\varphi(\boldsymbol{x})$ is an eigenfunction of $\mathcal{K}$ and a given double layer density on S. By dividing both equations, the second characteristic equation is obtained:

$$\kappa_n = \frac{\psi_n^{(e)}(\boldsymbol{\eta})}{\psi_n^{(i)}(\boldsymbol{\eta})} = \frac{\lambda_n - 1}{\lambda_n + 1} \quad . \tag{3.34}$$

By combining (3.31) and (3.34), we get the eigenvalues:

$$\lambda_n = \frac{1+\kappa_n}{1-\kappa_n} = -\frac{1}{2n+1} \quad ; n = 0, 1, 2, \ldots \; . \tag{3.35}$$

The range of the eigenvalues is $\lambda_n \in [-1, 0)$ and the *spectral radius* $\max_n |\lambda_n|$ is equal to 1. The only accumulation point of the eigenvalues is 0.

3.2.4 Nullspace

In this section we derive the null-space of the operator $1 + \mathcal{K}$:

$$(1+\mathcal{K})\varphi(\boldsymbol{x}) = \varphi(\boldsymbol{x}) + \oint_S K(\boldsymbol{x}, \boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = 0 \; . \tag{3.36}$$

This null-space and the eigenspace of the operator $\mathcal{K}$ for the eigenvalue -1 are identical, because

$$\mathcal{K}\varphi(\boldsymbol{x}) = -1 \cdot \varphi(\boldsymbol{x}) \; . \tag{3.37}$$

These two spaces play an important role in the subsequent sections.

First, we prove that the null-space $N(1+\mathcal{K})$ is non-trivial and second, that its dimension is exactly 1. The statement that the null-space is non-trivial is equivalent to $\dim[N(1+\mathcal{K})] \geq 1$. We consider the electric potential inside the particle $\boldsymbol{x} \in V_e$. But so far, we have transformed Laplace's equation only for the interior domain V and not for the exterior domain V_e. This transformation in the exterior domain is performed in the same way as the one for the interior

domain in Section 3.1.2. For the unknown double layer density $\varphi(\boldsymbol{x})$ on the particle surface S we obtain a boundary integral equation similar to Equation (3.14):

$$\frac{1}{2}\varphi(\boldsymbol{x}) = \oint_S G(\boldsymbol{x},\boldsymbol{\xi})\hat{\boldsymbol{n}}(\boldsymbol{\xi})\cdot\nabla_{\boldsymbol{\xi}}\varphi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} - \oint_S \nabla_{\boldsymbol{\xi}}G(\boldsymbol{x},\boldsymbol{\xi})\cdot\hat{\boldsymbol{n}}(\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}}\ . \tag{3.38}$$

Note the sign change compared to Equation (3.14). The domain of integration is now the inside of the particle and not its outside, but the surface normal $\hat{\boldsymbol{n}}$ is still pointing to the outside.

Let us assume that $\varphi(\boldsymbol{x}) = \text{const}; \boldsymbol{x} \notin \overline{V}$ is a null-function of $1+\mathcal{K}$. The single layer integral in Equation (3.38) vanishes, so that

$$\frac{1}{2}\varphi(\boldsymbol{x}) = -\frac{1}{2}\mathcal{K}\varphi(\boldsymbol{x})\ . \tag{3.39}$$

This is:

$$(1+\mathcal{K})\varphi(\boldsymbol{x}) = 0\ . \tag{3.40}$$

This is exactly the definition of the null-function. Thus $\varphi(\boldsymbol{x}) = \text{const}$ is one null-function of $1+\mathcal{K}$, so that the null-space of $1+\mathcal{K}$ is non-trivial and its dimension $\dim[N(1+\mathcal{K})] \geq 1$.

To prove that $\dim[N(1+\mathcal{K})] = 1$ we have to prove now that $\dim[N(1+\mathcal{K})] \leq 1$. Therefore, we consider the adjoint problem:

$$(1+\mathcal{K}^*)\sigma(\boldsymbol{x}) = 0 \quad ; \boldsymbol{x} \in S\ . \tag{3.41}$$

This is the normal derivative of the single layer potential on the inner side of the particle surface:

$$\begin{aligned} \lim_{\epsilon \to 0-} \left. \frac{\partial \psi(\boldsymbol{x})}{\partial \hat{\boldsymbol{n}}} \right|_{\boldsymbol{x}=\boldsymbol{\eta}+\epsilon\hat{\boldsymbol{n}}} &= \lim_{\epsilon \to 0-} \oint_S K^*(\boldsymbol{x}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \\ &= \oint_S K^*(\boldsymbol{\eta}, \boldsymbol{\xi}) \sigma(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} + \sigma(\boldsymbol{\eta}) \end{aligned} \tag{3.42}$$

using the jump condition of the adjoint operator (3.25). Using the same notation for the exterior potential as in Section 3.2.3 we have:

$$\frac{\partial \psi^{(e)}}{\partial \hat{\boldsymbol{n}}} = (1 + \mathcal{K}^*)\sigma = 0 \; . \tag{3.43}$$

The *energy dissipation rate* of the single layer potential in the exterior domain $\boldsymbol{x} \notin \overline{V}$ or $\boldsymbol{x} \in V_e$ is given by

$$D^{(e)} = \frac{1}{8\pi} \int_{V_e} \nabla \psi^{(e)} \cdot \nabla \psi^{(e)} \, dV = \frac{1}{8\pi} \oint_S \psi^{(e)} \frac{\partial \psi^{(e)}}{\partial \hat{\boldsymbol{n}}} \, dS \; . \tag{3.44}$$

Since $\frac{\partial \psi^{(e)}}{\partial \hat{\boldsymbol{n}}} = 0$ we also have $\nabla \psi^{(e)} = 0$ for all $\boldsymbol{x} \in V_e$, which implies that the single layer potential ψ is constant in V_e. The single layer potential ψ is continuous across the particle surface S, so ψ is also constant on the interior side of S. So we have a *Dirichlet boundary value problem* with the boundary condition $\psi(\boldsymbol{x}) = \psi^{(i)} = \psi^{(e)} = \text{const}$ in the interior region $\boldsymbol{x} \in V^0$ or $\boldsymbol{x} \notin V_e$. By the uniqueness theorem of solutions of Laplace's equation there is simply one degree of freedom. Thus $\dim[N(1 + \mathcal{K}^*)] \leq 1$.

The operators $\mathcal{K}$ and $1 + \mathcal{K}$ are defined in a *Banach space*, which is a normed linear real (or complex) space that is complete. Furthermore, our operator $\mathcal{K}$ is a completely continuous

linear operator in this space $\boldsymbol{x} \in V^0$, and the *Fredholm-Riesz-Schauder theory* [15] is applicable. One of its theorems states that the dimension of the null-spaces of both operators $1 + \mathcal{K}$ and $1 + \mathcal{K}^*$ are identical. Thus we also have $\dim[N(1 + \mathcal{K})] \leq 1$.

Since the constraint derived beforehand states that the dimension of $N(1+\mathcal{K})$ is also greater or equal to 1, the dimension is equal to 1 ($\dim[N(1+\mathcal{K})] = 1$) and its basis function is $\varphi(\boldsymbol{x}) = 1$. The normalization of this basis function with respect to the inner product

$$\langle f, g \rangle = \frac{1}{S} \oint\!\!\!\!\!\!\int_S f g \, dS \tag{3.45}$$

yields the normalized basis function of the null-space on the surface of body α:

$$\varphi^{(\alpha)}(\boldsymbol{x}) = \frac{1}{\sqrt{S_\alpha}} \, , \tag{3.46}$$

where S_α is the surface area of body α.

Since the dimension of the null-space is non-zero, the operator $1 + \mathcal{K}$ is singular and cannot be inverted. The Fredholm-Riesz-Schauder theory says that the deficiency in its range is equal to the dimension of its null-space[4]. Its range is completed in Section 3.4 which makes the operator $1 + \mathcal{K}$ amenable to inversion.

[4]This corresponds to the case of square matrices, where the column rank equals the row rank.

3.3 Completed Double Layer Boundary Integral Representation

This and the subsequent sections present a modified form of the integral equation (3.14), which can be solved very efficiently. The single layer potential in the integral equation (3.14) is replaced by a field created by a point charge of magnitude Q_α and position $\boldsymbol{x}_\alpha \in V_e$:

$$-\oint_S G(\boldsymbol{x},\boldsymbol{\xi})\hat{\boldsymbol{n}}(\boldsymbol{\xi})\cdot\nabla_{\boldsymbol{\xi}}\psi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} \longrightarrow \frac{Q_\alpha}{|\boldsymbol{x}-\boldsymbol{x}_\alpha|}\ . \tag{3.47}$$

The position of the point charge inside the perfectly conducting body is arbitrary. However, numerical considerations can help to find an optimal position, for which the numerical solution is most accurate and the convergence time is optimal [41]. This optimal position depends on the particle configurations considered and the given shape of the bodies, so that there is no simple approach to this optimal position.

In our problem, Q_α, the net charge of particle α, is given and the constant potential ψ_α on the particle surface is sought[5]. The modified integral representation, which we call the *completed double layer boundary integral representation*, is

$$\psi(\boldsymbol{x}) = \psi^\infty(\boldsymbol{x}) + \frac{Q_\alpha}{|\boldsymbol{x}-\boldsymbol{x}_\alpha|} + \oint_S K(\boldsymbol{x},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} \quad ;\boldsymbol{x}\in V^0 \tag{3.48}$$

where $\varphi(\boldsymbol{\eta})\ ;\boldsymbol{\eta}\in S$ is the *double layer density* and K is the double layer kernel as defined in Equation (3.16). Note that the double layer integral does not represent the entire electric field

[5]This is the *elastance* problem, i.e., the inverse of the *capacitance* problem.

created by a charged body. For this reason, the potential field of a point charge has been added in Equation (3.48). By using the jump condition (3.18), an integral equation for the unknown double layer density φ is obtained on the particle surface S.

$$\psi_\alpha = \psi(\boldsymbol{\eta}) = \psi^\infty(\boldsymbol{\eta}) + \frac{Q_\alpha}{|\boldsymbol{\eta} - \boldsymbol{x}_\alpha|} + \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} + \varphi(\boldsymbol{\eta}) \quad ; \boldsymbol{\eta} \in S\ . \tag{3.49}$$

This is a *Fredholm equation of the second kind*, since the double layer density φ also appears outside the integral. The advantage of this equation is that upon discretization the linear algebraic system is well-conditioned, which is not the case for a *Fredholm equation of the first kind* like Equation (3.14).

3.4 Wielandt's Deflation

The operator $1+\mathcal{K}$ in Equation (3.49) has an incomplete range, since its null-space has dimension 1 as shown in Section 3.2.4. For this reason, $1 + \mathcal{K}$ is singular and cannot be inverted. The range of $1 + \mathcal{K}$ is completed by assigning the unknown potential to a projection of the double layer density to the null-space:

$$\begin{aligned} \psi_\alpha &= -\varphi^{(\alpha)} \left\langle \varphi, \varphi^{(\alpha)} \right\rangle \\ &= -\frac{1}{S} \oint_S \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \end{aligned} \tag{3.50}$$

where $\varphi^{(\alpha)}$ is the normalized basis function of the null-space. The choice of the orthogonality conditions is somewhat arbitrary. However, in order to remove the indeterminacy, the projections of the chosen vectors to the null-space must be linearly independent, i.e., the volume of the corresponding parallelepiped in the null-space must be nonzero. This volume being small although nonzero can lead to numerical difficulties through an ill-conditioned linear system. As a good choice, the null-function of $1 + \mathcal{K}$ is used.

Equation (3.50) is a further condition for the double layer density $\varphi^{(\alpha)}$ and is inserted into Equation (3.49). The resulting equation for the unknown double layer density on the particle surface S is:

$$\varphi(\boldsymbol{\eta}) = -\psi^{\infty}(\boldsymbol{\eta}) - \frac{Q_\alpha}{|\boldsymbol{\eta} - \boldsymbol{x}_\alpha|} - \frac{1}{S}\oint_S \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} - \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \ . \tag{3.51}$$

In general, the term *deflation* refers to algorithms for moving eigenvalues to the origin (see Figure 3.1). Wielandt's deflation [31] uses only the eigenfunctions of the operator and not those of its adjoint operator. For our problem, we know the eigenfunction corresponding to the eigenvalue $\lambda = -1$ of the operator $\mathcal{K}$, since this eigenfunction is identical to the basis of the null-space of the operator $1 + \mathcal{K}$, as discussed in Section 3.2.4. This eigenvalue is deflated to 0, which, in the case of one sphere, reduces the spectral radius to $\frac{1}{3}$, the absolute value of the next dominant eigenvalue of $\mathcal{K}$. As mentioned beforehand, $\mathcal{K}$ has just one accumulation point of eigenvalues, namely 0. Since 1 is not an accumulation point, the spectral radius is strictly reduced and relaxation methods with fast convergence can be applied.

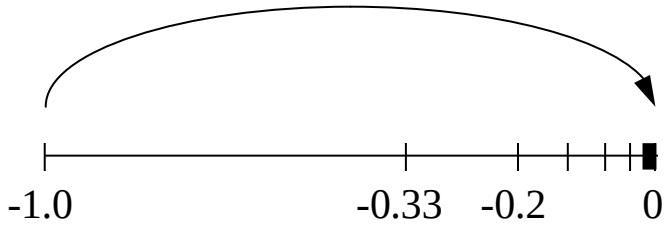

Figure 3.1: Wielandt's deflation moves the eigenvalue at -1 to the accumulation point 0.

In Equation (3.51) the operator $\mathcal{K}$ is replaced by the new operator

$$\mathcal{H} = \mathcal{K} + \varphi^{(\alpha)} \left\langle \bullet, \varphi^{(\alpha)} \right\rangle \ . \tag{3.52}$$

Note that $\varphi^{(\alpha)}$ is still an eigenfunction of $\mathcal{H}$, but now its corresponding eigenvalue is 0.

Wielandt's deflation can be illustrated for matrices in canonical form:

$$\begin{pmatrix} -1 & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{3} & 0 & \cdots & 0 \\ 0 & 0 & -\frac{1}{5} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} + \varphi^{(\alpha)} \left\langle \bullet, \varphi^{(\alpha)} \right\rangle = \begin{pmatrix} 0 & X & X & \cdots & X \\ 0 & -\frac{1}{3} & 0 & \cdots & 0 \\ 0 & 0 & -\frac{1}{5} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} . \tag{3.53}$$

Wielandt's deflation destroys the canonical form of the matrix, but the matrix remains triangular. The eigenvalues of a triangular matrix can be read off the diagonal. Just one eigenvalue has been shifted to 0, the remaining eigenvalues have not been shifted. Here the canonical

form is used for illustration, but the results also apply to the original system, since similarity transformations applied to obtain the canonical form preserve the eigenvalues.

3.5 Multiparticle Problem

The equations derived so far are valid for a single body α in an infinite unbounded space. In this section the equations are generalized to N bodies in an infinite unbounded space. Each body k is described by its associated point charge of magnitude Q_k, its position $\boldsymbol{x}_k$, and its surface S_k. We have N bodies, so k varies in the range from 1 to N. Now the boundary surface S of the vacuum in the *completed double layer boundary integral representation* (3.48) is the union of all body surfaces S_k. Instead of having just one point charge, we have an array of N point charges, which create a sum of potential fields. These ideas are expressed in the following equation corresponding to the equation for one body (3.48):

$$\psi(\boldsymbol{x}) = \psi^{\infty}(\boldsymbol{x}) + \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{x} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{x}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \right\} \quad ; \boldsymbol{x} \in V^0 \,. \tag{3.54}$$

By using the jump condition (3.18), we obtain the boundary integral equation on the surface S_k of particle k corresponding to Equation (3.49):

$$\psi_k = \psi(\boldsymbol{\eta}) = \psi^{\infty}(\boldsymbol{\eta}) + \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{\eta} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \right\} + \varphi(\boldsymbol{\eta}) \quad ; \boldsymbol{\eta} \in S_k \tag{3.55}$$

where ψ_k is the uniform potential on the perfectly conducting body k.

Now we have a new operator $1+\mathcal{K}$ with a new null-space. This null-space has one dimension for each body surface and is altogether of dimension N and is spanned by N basis functions. There are N distinct basis functions, one for each body surface:

$$\varphi^{(k)}(\boldsymbol{x}) = \begin{cases} \frac{1}{\sqrt{S_k}} & ; \boldsymbol{x} \in S_k \\ 0 & ; \boldsymbol{x} \notin S_k \end{cases} \qquad ; k = 1...N \ . \tag{3.56}$$

After Wielandt's deflation has been applied in parallel for all basis functions of the null-space, the following integral equation for the unknown double layer density φ on the different particle surfaces S_k is obtained:

$$\varphi(\boldsymbol{\eta}) = -\psi^{\infty}(\boldsymbol{\eta}) - \frac{1}{S_k} \oint_{S_k} \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} - \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{\eta} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \right\} \tag{3.57}$$

$$\boldsymbol{\eta} \in S_k \qquad ; k = 1...N \ .$$

This system of integral equations cannot be solved analytically for arbitrary body shapes and number of bodies. The rest of this work approaches the solution of this system numerically, using boundary elements and fixed-point relaxation methods. Having obtained the solution φ of the equation system (3.57), the potential on each body k can be calculated using Equation (3.50):

$$\psi_k = -\frac{1}{S_k} \oint_{S_k} \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \ .$$

The potential in the vacuum is readily available by evaluating Equation (3.54).

Chapter 4

Numerical Approach

There are two steps involved in the numerical discretization of Equation (3.57). First, the double layer density φ has to be approximated numerically, based on a finite set of variables. Second, the integrals have to be evaluated with a finite computable approximation.

The body surfaces are divided into similarly shaped *boundary elements*. For boundary elements we use planar triangles, which are sufficient for the considered case of spherical bodies. However, to solve problems with more complex bodies, these planar boundary elements should be replaced by more efficient curved boundary elements.

The integrals are evaluated on each boundary element separately using *Gauss-Legendre quadrature*. The following sections describe the boundary elements and the quadrature. The result of this chapter is a linear algebraic equation system that is solved by relaxation methods. This chapter mentions 'iteration' and 'iterations steps' frequently. Please refer to Chapter 5 for an explanation of these terms.

4.1 Boundary Elements

Each body surface is divided into M boundary elements, which are similarly sized planar triangles in our case. The vertices of these triangles are elements of the bodies' surfaces. A boundary element is described by an isoparametric formulation using interpolation functions. The following transformation transforms the 2-dimensional natural coordinate system with the coordinates (r, s) to the 3-dimensional coordinate system $\boldsymbol{x}$:

$$\boldsymbol{x} = \sum_{i=1}^{q} h_i(r, s)\boldsymbol{p}_i \ . \tag{4.1}$$

Here $\boldsymbol{p}_i$ are points on one boundary element's edges and h_i are interpolation functions, also known as shape functions. Since we use planar triangles, q is equal to 3 and the transformation becomes:

$$\boldsymbol{x} = \boldsymbol{p}_1 + r(\boldsymbol{p}_2 - \boldsymbol{p}_1) + s(\boldsymbol{p}_3 - \boldsymbol{p}_1) \tag{4.2}$$

where $\boldsymbol{p}_i$ are the triangle's vertices. The natural coordinates vary in the ranges $0 \leq r \leq 1$ and $0 \leq s \leq 1 - r$. The triangle vertex $\boldsymbol{p}_1$ corresponds to $(r, s) = (0, 0)$, $\boldsymbol{p}_2$ corresponds to $(r, s) = (1, 0)$, and $\boldsymbol{p}_3$ corresponds to $(r, s) = (0, 1)$. This range spans one whole boundary element and the integration on one boundary element is carried out over this range.

To obtain a polyhedron with an arbitrary number of planar triangles, we use an algorithm called *tesselation*. We start with a polyhedron with a fixed number of triangles — a tetrahedron, a octahedron, or an icosahedron, for instance. Then each triangle is divided into 4 new equally

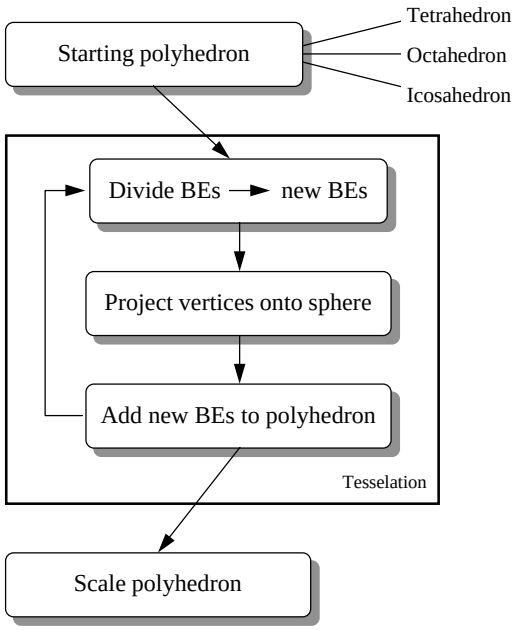

Figure 4.1: Tesselation.

sized triangles, whose vertices are the vertices of the old triangle plus the center points of the edges of the old triangle. Since these center points are not on the charge's surface, this procedure results in an error. This error can be omitted by projecting these center points onto the charge's surface, thus all the new triangle vertices are also on the sphere's surface. This iteration step which is presented in Figure 4.1 is run iteratively, so that polyhedra of arbitrary order can be obtained. As a further refinement, we scale the resulting polyhedra, so that its surface is equal to the sphere surface. Its vertices are not on the sphere's surface but outside the sphere. Figure 4.2, Figure 4.3, and Figure 4.4 show polyhedra with 20, 80, and 320 elements.

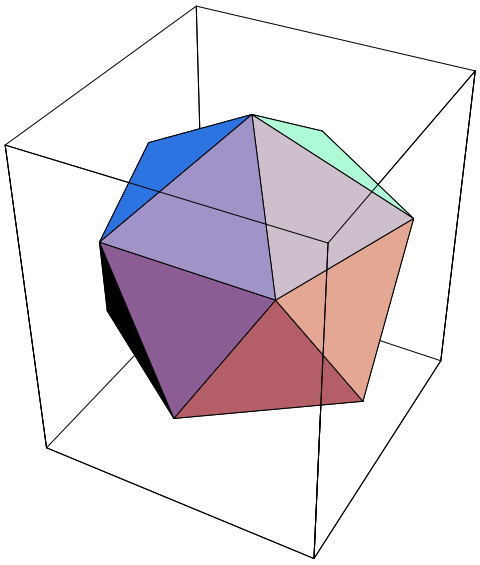

Figure 4.2: Polyhedron with 20 elements (icosahedron).

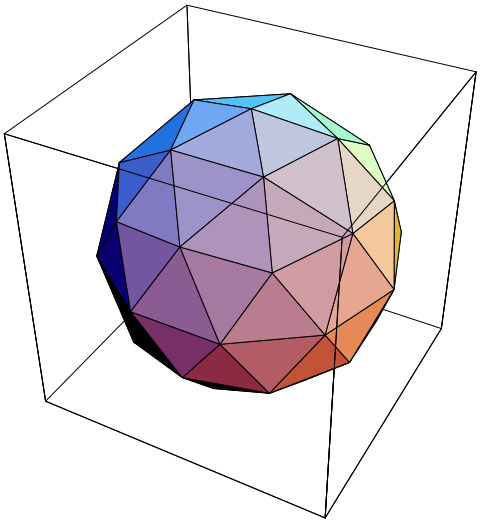

Figure 4.3: Polyhedron with 80 elements.

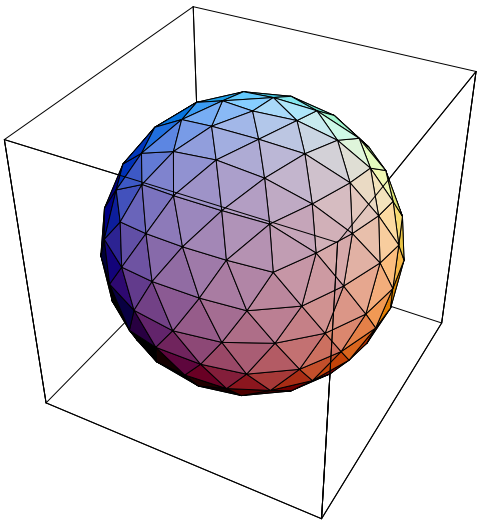

Figure 4.4: Polyhedron with 320 elements.

4.2 Numerical Quadrature

Collocation is used to reduce the infinite domain of the double layer density $\varphi(\boldsymbol{\eta})$ to a finite set of variables at distinct points $\varphi(\boldsymbol{\eta}_{ki}) = \varphi_{ki}$[1]. The collocation ensures that the error of φ vanishes at these collocation points $\boldsymbol{\eta}_{ki}$. In our approach to the problem, one boundary element has one collocation point, its centroid, so that the double layer density $\varphi(\boldsymbol{\eta})$ on the whole boundary element S_{ki} is equal to its value at the one collocation point:

$$\varphi(\boldsymbol{\eta}) = \varphi(\boldsymbol{\eta}_{ki}) \quad ; \boldsymbol{\eta} \in S_{ki} \ . \tag{4.3}$$

Hence the double layer density is assumed to be constant on one boundary element. This approximation can be refined using collocation of higher order. However, first order collocation is sufficient for our spheres problems and leads to sufficiently accurate solutions, as shown in Section 7.1. The major advantage of using first order collocation is to reduce the number of unknowns φ_{ki}: there is only one unknown per boundary element. This leads to less computation in one iteration step and to less memory usage. The drawback is that the decrease in the error per iteration step is not as big as it would be in the case of higher order collocation and the final converged solution is not as accurate. However, the less computation allows us to run more iteration steps in the same time period and the less memory usage allows us to use more boundary elements, which partly compensates for the loss of accuracy.

The integration is approximated by Gauss-Legendre quadrature. In contrast to classical

[1]The index ki denotes the ith boundary element on the kth body's surface.

integration formulas, such as Simpson's rule, where the integral of a function is approximated by the sum of its functional values at a set of equally spaced points, Gaussian quadrature formulas choose collocation points which are not necessarily equally spaced. In addition to choosing the weight coefficients, there is a further degree of freedom in choosing these points on the abscissa, which leads to a higher order of approximation using the same number of functional evaluations. Gaussian quadrature formulas are all of the following type:

$$\int_{x=a}^{b} W(x)f(x)\,dx \approx \sum_{r=1}^{Q} w_r f(x_r)\ . \tag{4.4}$$

Different functions $W(x)$ determine different classes of the Gaussian quadrature and can be chosen to remove integrable singularities from the desired integral. The most common chosen $W(x)$ is $W(x) = 1$, which leads to the Gauss-Legendre quadrature. The weight coefficients w_r and the collocation points x_r are chosen, so that the residual

$$\int_{x=a}^{b} W(x)f(x)\,dx - \sum_{r=1}^{Q} w_r f(x_r) \tag{4.5}$$

is minimal. The theory behind Gaussian quadratures is closely tied to the theory of orthogonal polynomials. The function $f(x)$ is approximated by polynomials, which have the same functional values as $f(x)$ at the Q collocation points x_r. This theory is well described and the values of w_r and x_r are listed in [47].

In our case, we have a 2-dimensional integration to evaluate instead of a 1-dimensional one, as described in Formula (4.4). A common way to evaluate this integration is to divide the

integration domain into a 2-dimensional grid of lines parallel to both coordinate axes spanning one boundary element. The offsets of these straight lines are the collocation points of the 1-dimensional Gaussian quadrature. First, a 1-dimensional Gaussian quadrature is executed along the lines parallel to the first coordinate axis which leads to a 1-dimensional set of functional values. Second, a further Gaussian quadrature is executed along the second coordinate axis using those functional values. For the 2-dimensional integration of a scalar function $g(x, y)$ with a 2-dimensional input set over a rectangular domain spanned by a cartesian coordinate system, the numerical integration is as follows:

$$\int_{y=c}^{d} \int_{x=a}^{b} g(x, y)\, dx\, dy \approx \sum_{r=1}^{Q} w_r \sum_{s=1}^{Q} \tilde{w}_s g(x_r, \tilde{x}_s) \ . \tag{4.6}$$

Note that, if the weighting coefficients and collocation points along the x-axis, w_r and x_r, are given, we have to scale the corresponding $\tilde{w}_s$ and $\tilde{x}_s$ for the y-axis in the following way:

$$\tilde{w}_s = \frac{d-c}{b-a} w_s \quad ; s = 1...Q \ , \tag{4.7}$$

$$\tilde{x}_s = \frac{d-c}{b-a} x_s \quad ; s = 1...Q \ . \tag{4.8}$$

However, our boundary elements are no rectangles but triangles. The quadrature formula given in Equation (4.6) is modified to evaluate an integral over a triangle with the vertices $\boldsymbol{p}_1$,

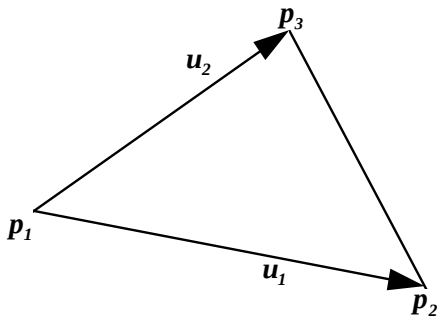

Figure 4.5: Boundary element. The boundary element is a planar triangle with the vertices $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$.

$\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. The vectors describing its edges are

$$\boldsymbol{u}_1 = \boldsymbol{p}_2 - \boldsymbol{p}_1 \ , \tag{4.9}$$

$$\boldsymbol{u}_2 = \boldsymbol{p}_3 - \boldsymbol{p}_1 \ , \tag{4.10}$$

as shown in Figure 4.5. The scalar function $h(\boldsymbol{x})$ is defined on this triangle. According to the isoparametric formulation, the transformation from the 2-dimensional natural coordinates (r, s) to the 3-dimensional system $\boldsymbol{x}$ is given by Equation (4.2). The integral of $h(\boldsymbol{x})$ over the triangular area is given by:

$$\int_{S_\triangle} h(\boldsymbol{x})\, dS_x = \int_{r=0}^{1} \int_{s=0}^{1-r} h(\boldsymbol{p}_1 + \boldsymbol{u}_1 r + \boldsymbol{u}_2 s)\, |\boldsymbol{u}_1 \times \boldsymbol{u}_2|\ ds\, dr \tag{4.11}$$

where $|\boldsymbol{u}_1 \times \boldsymbol{u}_2|$ is the Jacobian of the transformation. According to the lines of the 2-dimen-

sional quadrature for a rectangle, the following numerical approximation for the integral on the triangle is obtained:

$$\int_{S_\triangle} h(\boldsymbol{x})\, dS_{\boldsymbol{x}} \approx |\boldsymbol{u}_1 \times \boldsymbol{u}_2| \sum_{r=1}^{Q} w_r \sum_{s=1}^{Q} \tilde{w}_{rs} h(\boldsymbol{p}_1 + \boldsymbol{u}_1 x_r + \boldsymbol{u}_2 \tilde{x}_{rs}) \tag{4.12}$$

where

$$\tilde{w}_{rs} = (1 - x_r) w_s \quad ; r = 1...Q, s = 1...Q \ , \tag{4.13}$$

$$\tilde{x}_{rs} = (1 - x_r) x_s \quad ; r = 1...Q, s = 1...Q \ . \tag{4.14}$$

In the boundary integral equation (3.57) the integral

$$\oint_{S_k} \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} = \sum_{j=1}^{M} \oint_{S_{kj}} \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \tag{4.15}$$

is approximated by

$$\oint_{S_k} \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \approx \sum_{j=1}^{M} \varphi_{kj} S_{kj} \ , \tag{4.16}$$

since the double layer density $\varphi(\boldsymbol{\xi})$ is constant on one boundary element $\boldsymbol{\xi} \in S_{kj}$[2].

The second integral is evaluated by Gauss-Legendre quadrature. The double layer integral over the whole body surface is equal to the sum of all integrals over the different boundary

[2] S_{kj} denotes both the domain and the area of the boundary element.

elements on the body surface:

$$\oint_{S_l} K(\boldsymbol{\eta},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} = \sum_{j=1}^{M} \oint_{S_{lj}} K(\boldsymbol{\eta},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} \ . \tag{4.17}$$

The integrand on the right-hand side is the product of the constant $\varphi(\boldsymbol{\xi})$ and the double layer kernel $K(\boldsymbol{\eta},\boldsymbol{\xi})$, which is not assumed to be constant on the boundary element. Hence $\varphi(\boldsymbol{\xi})$ can be moved out of the integral, whereas the remaining integral is solved by numerical quadrature and Equation (4.12) is applied with $h(\boldsymbol{x}) = K(\boldsymbol{\eta},\boldsymbol{x})$:

$$\oint_{S_{lj}} K(\boldsymbol{\eta},\boldsymbol{\xi})\,dS_{\boldsymbol{\xi}} \approx |\boldsymbol{u}_{lj,1} \times \boldsymbol{u}_{lj,2}| \sum_{r=1}^{Q} w_r \sum_{s=1}^{Q} \tilde{w}_{rs} K(\boldsymbol{\eta}, \boldsymbol{p}_{lj,1} + \boldsymbol{u}_{lj,1} x_r + \boldsymbol{u}_{lj,2} \tilde{x}_{rs}) \ . \tag{4.18}$$

In the case of spherical bodies, a natural position for the point charge $\boldsymbol{x}_k$ is the center of the sphere k. This might not be the optimal position for the point charge as discussed in Section 3.3, but is a very convenient approach for arbitrary configuration of spheres.

The final numerical approximation of the boundary integral equation (3.57) is:

$$\varphi_{ki} = \underbrace{-\psi^{\infty}(\boldsymbol{\eta}_{ki}) - \sum_{l=1}^{N} \frac{Q_l}{|\boldsymbol{\eta}_{ki} - \boldsymbol{x}_l|}}_{=b_{ki}} + \tag{4.19}$$

$$\underbrace{\left(-\frac{1}{S_k} \sum_{j=1}^{M} S_{kj}\varphi_{kj} - \sum_{l=1}^{N}\sum_{j=1}^{M} |\boldsymbol{u}_{lj,1} \times \boldsymbol{u}_{lj,2}| \sum_{r=1}^{Q} w_r \sum_{s=1}^{Q} \tilde{w}_{rs} K(\boldsymbol{\eta}_{ki}, \boldsymbol{p}_{lj,1} + \boldsymbol{u}_{lj,1} x_r + \boldsymbol{u}_{lj,2} \tilde{x}_{rs}) \varphi_{lj} \right)}_{=\sum_{l=1}^{N}\sum_{j=1}^{M} A_{kl\,ij}\varphi_{lj}}$$

$$k = 1 \ldots N \quad , \quad i = 1 \ldots M \ .$$

Note that $\psi^{\infty}(\boldsymbol{\eta})$ is also collocated at the collocation points $\boldsymbol{\eta} = \boldsymbol{\eta}_{ki}$. The discretized Equation (4.19) is a linear algebraic equation system:

$$\boldsymbol{x} = \tilde{\boldsymbol{A}} \cdot \boldsymbol{x} + \tilde{\boldsymbol{b}} \tag{4.20}$$

or

$$x_m = \sum_{n=1}^{NM} \tilde{A}_{mn} x_n + \tilde{b}_m \quad ; m = 1...NM \tag{4.21}$$

where the unknown vector is $x_{(k-1)M+i} = \varphi_{ki}$. The procedure in Chapter 3 guarantees that this equation system has one unique solution $\boldsymbol{x}$. The subsequent chapters of this thesis describe parallel relaxation methods to solve this system.

4.3 Widely Separated Boundary Elements

The calculation of the 2-dimensional Q-point quadrature described in Section 4.2 is very expensive. Furthermore, it yields an accuracy in the approximation of the integral that is not needed for far apart boundary elements ki and lj. Two boundary elements are considered to be far apart, if they neither belong to the same body surface nor to the surfaces of two neighboured bodies. In this case, instead of a quadrature with $Q > 1$ collocation points, a quadrature with one collocation point is used. The resulting discretized equation according to the lines of

Equation (4.19) is:

$$\begin{aligned}
\varphi_{ki} &= \underbrace{-\psi^{\infty}(\boldsymbol{\eta}_{ki}) - \sum_{l=1}^{N} \frac{Q_l}{|\boldsymbol{\eta}_{ki} - \boldsymbol{x}_l|}}_{=b_{ki}} \\
&+ \underbrace{\left(-\frac{1}{S_k} \sum_{j=1}^{M} S_{kj}\varphi_{kj} - \sum_{l=1}^{N}\sum_{j=1}^{M} S_{lj} K(\boldsymbol{\eta}_{ki}, \boldsymbol{\eta}_{lj})\varphi_{lj} \right)}_{=\sum_{l=1}^{N}\sum_{j=1}^{M} A_{kl\,ij}\varphi_{lj}} \qquad (4.22) \\
& k = 1...N \quad , \quad i = 1...M \ .
\end{aligned}$$

The algorithm to solve the linear algebraic equation system (4.19) or (4.22) is divided into two parts: the first part creates the matrix $\tilde{\boldsymbol{A}}$ and the vector $\tilde{\boldsymbol{b}}$ using the geometry of the problem, which basically involves calculating the double layer kernel K for each boundary element - boundary element pair. The second part solves this system of equations using fixed-point iteration methods.

Chapter 5

Parallel Computing

The linear algebraic equation system (4.19) or (4.22) derived in Sections 3 and 4 has a large number of unknowns and a huge system matrix $\boldsymbol{A}$. For a problem of N bodies, which are discretized into M boundary elements each, the number of unknowns is $M \cdot N$. The number of elements in $\boldsymbol{A}$ is $(M \cdot N)^2$. For instance, for $N = 1024$ and $M = 320$ the number of unknowns is $327,680$, which using single precision values is equivalent to 1 MB of memory, but even this is dwarfed by the 400 GB needed for the system matrix. The problem size and the large amount of computation demand the use of parallel computers.

The equation system (4.19) or (4.22) is solved by a parallel form of the *Jacobi iteration* that can be applied because of the small spectral radius of the system matrix. The Jacobi iteration can be easily parallelized, that is the equation system is split into smaller parts which can be solved on different processors in parallel. This reduces the runtime to a tolerable amount. A further advantage of the Jacobi iteration is that the system matrix is split into square block

matrices. To reduce the memory space just a part of these block matrices is stored permanently in memory, the rest reconstructed during the iteration. Furthermore, this parallel algorithm is modified to a *block Gauss-Seidel iteration*, which naturally fits our physical problem and reduces the execution time by an order of magnitude.

5.1 Parallel Computers

This section gives a general overview of parallel computers and discusses several aspects in parallel computing.

To implement an algorithm on a parallel computer, the algorithm has to be *parallelized*. This procedure of parallelization has several issues that do not occur in the serial implementation of this algorithm. The first issue is splitting the total workload of the algorithm into smaller tasks distributed to different processes, which is known as *task allocation*. The second issue is *communication* between these processes. Communication is expensive on parallel machines compared to pure computation: a memory or network access is to two orders of magnitude slower than a floating-point operation and this discrepancy will most likely increase on future machines. A third issue is the *synchronization* of the task execution on the different processors. Synchronization is necessary if the processes are dependent on each other, or if they exchange data. In general, one such process which executes one part of the overall task, is run on one processor.

Because of these different issues, there is a whole set of different parallel computers. Depending on their characteristics, parallel computers can be classified in the following classes:

Number of Processors. Parallel machines can be distinguished by number of processors and communication granularity. Machines are called *massively-parallel*, if they have hundreds to thousands of processors. Communication granularity specifies the overhead of communicating relative to computation (for example, a floating-point operation). A machine has *coarse-grain* communication when communication takes hundreds to thousands of floating-point operation times, since the communication must initiate work taking thousands of floating-point operation times to reasonably amortize communication overhead. A machine has fine-grain communication when communication takes tens of floating-point operation times.[1]

It remains to be seen, however, how fine-grained the communication of future massively-parallel machines will be. Massively-parallel machines would like to have fine-grain communication so that they can divide the work on a parallel job across many processors, but physical contraints place more processors further apart, favoring more coarse-grained communication.

Global Control Mechanism. All parallel computers have some kind of a global control mechanism. They are classified by what extent their processors are controlled by this mechanism. There are two different main kinds of parallel computers:

- *SIMD = Single Instruction Multiple Data.* The different processors of a SIMD machine run the same sequence of instructions synchronously. Each processor runs instructions on a different set of data, so that the data set is distributed to the different processors. SIMD machines' power lies in broadcasting, where distributed data is collected or parallel

[1] This is not currently a concensus on the exact boundaries between fine- and coarse-grain.

operations are applied to distributed data in a very efficient way.

- *MIMD = Multiple Instruction Multiple Data.* These computers not only have different data sets, but also execute different sequences of instructions on different processors independently. The different program versions presented in this thesis are all *SPMD = Single Program Multiple Data* codes, which is a common programming model for both SIMD and MIMD machines. Each processor runs the same program, but in the case of the MIMD model it can control its instruction flow by control directives, like `if` or `for`.

Programming Models. This paragraph is also covered in our paper [51]. A programming model is an abstraction from both programming languages and computer hardware that specifies the operations that may be performed and the cost of performing them without going into too much detail. For instance, single-processor computers are unified by the *von Neumann* programming model that allows people to switch languages (e.g., FORTRAN to C) and computers (VAX to DECstation) without great disruption. The von Neumann model's consensus is so broad that many users forget it is there.

Broadly speaking, today's parallel computers support three programming models: *message-passing, shared memory,* and *data parallel.* Our program has been implemented in the former two programming models.

In the *message-passing programming model* each of the P processors has its own local address space. A program in this model uses these P address spaces and communicates between the processors by explicitly sending messages. An *address space* is a mapping from addresses

(variable names in source codes) to memory locations (storage of variable values). Thus, on a message-passing computer, the hardware will treat accesses to one variable by two different processors as referring to two different locations. The cost model for message-passing is that messages have a large start-up cost, implying that performance will be better for programs that send only a few, large messages.

The *shared-memory programming model* uses P processors, but allows them to share a single address space. Thus, accesses to the same address automatically refer to the same variable. A consequence of this variable sharing is that communication occurs implicitly whenever a processor reads a location last written by another.

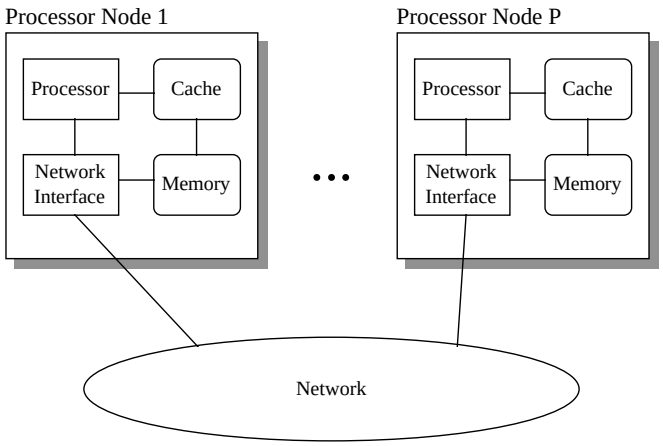

Figure 5.1: Organization of a parallel computer. Many parallel computers are composed of P processor nodes, each of which contains a microprocessor, cache, and physically local memory. The nodes are connected by an interconnection network (e.g., a 2D mesh or a fat tree).

A common myth regarding shared memory is that all memory must reside in a central place. To the contrary, most future shared-memory computers will use the same physically-

distributed memory modules as a message-passing computer (see Figure 5.1). In contrast to message-passing, however, shared-memory programmers and compilers do not have to know exactly where data reside, because they to not have to move it between address spaces with explicit messages.

The algorithm described in the upcoming sections has been implemented as programs in three different programming environments:

- *CMMD message-passing library.* The program is implemented in ANSI-C on a Thinking Machines CM-5 and calls CMMD library functions [49]. The program's data set is distributed to the local address spaces of the different processors. Data between these processors is exchanged by sending messages on the data network, which connects the processors of the CM-5 in a fat tree structure [48].

- *Split-C.* Split-C is a multi-processing programming language developed by the Computer Science Devision at the University of California - Berkeley [11, 12]. The programming model of Split-C is the static shared-memory model. The address space in Split-C is shared and 2-dimensional, where the first coordinate is the processor number and the second the local address on this processor. A memory access is called *local*, if the accessing processor p accesses a memory location with its first coordinate equal to p. This memory location is locally stored on processor p. If the memory location has a first coordinate different from p, the memory access is called *remote*. By using *global pointers* and issuing *split-phase assignments* or *signaling stores*, all processors can access the shared 2-dimensional address space. The cost model for Split-C is similar to the message-passing model, where local

memory accesses are fast and cheap, but remote memory accesses are slow and expensive. Split-C creates a communication protocol that uses communication primitives called *active messages*, which closely match the hardware functionality of the CM-5. However, the implementation of these primitives in the current Version 1.0 of Split-C is based on *polling*[2] and does not take use of the CM-5's hardware interrupts. For this reason, the Split-C program has many similarities to the message-passing program, where receiving messages is also based on polling.

- *Cooperative shared memory (CSM).* The programming model of CSM [24] is the dynamic shared-memory model, where accessed remote data is automatically copied to the local *cache* of the accessing processor. A cache maintains a high-speed memory buffer closely coupled to a processor, and copies memory locations into this buffer, which are likely to be accessed in one of the subsequent instructions issued by the processor [23]. Since accesses to a cache are much faster than memory accesses, especially in the case of remote memory accesses of parallel computers, instruction sequences, which access the same memory locations frequently and therefore have a high cache hit-ratio, execute much faster on computers with caches than on computers without caches. The implementation of CSM in the *Check-In / Check-Out (CICO) model* provides a cost model, where accesses to the local caches are cheap, but accesses to memory locations in shared memory are expensive, if they are not copied to the local cache of the accessing processor.

[2]The term *polling* states that a processor has to check its input buffer, in order to see if it has received any messages.

5.2 Parallel Programming

5.2.1 Performance Issues

The expressions and methods as already discussed in [5, 17] and summarized in this section provide information about the efficiency and quality of a parallel program. These methods are necessary to predict the performance of a program on future large-scale parallel machines. For example, we want to simulate ER fluids with up to one million particles. The behavior of a program on a parallel machine with respect to numbers of processors or different input set sizes is widely known as *scaling*. The different methods to test the scaling behaviour are called *scaling models*.

One of these methods, *constant-problem-size* scaling, which is repeatedly used in this work, assumes that the program parameters are not changed at all during the scaling studies. The same program with a constant input set is run on a parallel machine several times using different numbers of processors. Using P processors and a problem of size N (number of bodies, for example), we get the first feature for the quality of a parallel program:

$$S_P(N) = \frac{T^*(N)}{T_P(N)} \tag{5.1}$$

is the *speedup* of the program. $T^*(N)$ is the execution time of the optimal serial (uniprocessor) version to solve the problem of size N. $T_P(N)$ is the execution time of the parallel version with P processors. The closer $S_P(N)$ gets to the ideal relation $S_P(N) = P$, the better is the parallel program. In general, this ideal line is never reached.

Another magnitude, having the same information as the speedup, is the ratio

$$E_P(N) = \frac{S_P(N)}{P} = \frac{T^*(N)}{PT_P(N)} \tag{5.2}$$

called *efficiency*. In the ideal case $E_P(N)$ is equal to 1.

There is a practical difficulty with the definitions given above because the optimal serial time $T^*(N)$ is unknown in general. For this reason, there are differently defined alternatives for $T^*(N)$, as follows:

- $T^*(N)$ is the execution time of the best existing serial version.
- $T^*(N)$ is the execution time on the parallel machine with one processor. In the case of constant-problem-size scaling, it is not efficient or not possible to obtain $T^*(N)$ with one processor. The problem would have to be small enough to be solved on one processor. But by distributing this problem onto many processors, the workloads of the processors might get too small without any significant statement about the scaling. Thus, to keep the workloads high, larger problems have to be chosen, which cannot be solved on one processor because they exceed its resources.
- $T^*(N)$ is the execution time of the program on the parallel machine with P^* processors, where $P^* \leq P$.[3] The speedup based on this execution time does not include any penalties caused by the transformation of the sequential program into the parallel program, but

[3] Speedup studies based on this number are widely used throughout this work.

yields significant results for the scaling of the parallel program because of sufficiently high workloads.

The latter two alternatives do not provide any information on the absolute merits of the algorithm in contrast to the original definition of $T^*(N)$. However, this can be seen as an advantage, since they provide direct information on the quality of the parallelization.

A fundamental issue is whether the maximum attainable speedup $S_\infty(N)$ can be made arbitrarily large, as N is increased. The main difficulty is that, in general, programs have sections that are inherently sequential, such as input or output subroutines. Those sections become bottlenecks in the highly parallelized version of the program. This is known as *Amdahl's law* [3] and can be quantified as follows: if a program consists of two sections, one that is inherently sequential and the other that is fully parallelizable, and if the sequential section consumes a fraction f of the total execution time, the attainable speedup is bounded by:

$$S_p(N) \leq \frac{1}{f + (1-f)/P} \leq \frac{1}{f} \qquad ; \forall P \; . \tag{5.3}$$

Note that, for computational problems for which f tends to zero as the size of the problem N increases, Amdahl's law is not a concern.

Another method for testing the scaling of a parallel program used in this work is the *time-constrained* scaling. Hereby not the input set is kept constant, but the execution time. An *a priori* execution time for the program is modeled as a function of the number of processors and the input set. Then the program is run on different numbers of processors with different input

sets, so that the *a priori* estimated execution time is kept constant. The actual execution times T_P are all the same in the ideal case, but in reality they obey the following constraint:

$$T_P \geq T_{P-1} \quad ; P > 1 \tag{5.4}$$

where P denotes the number of processors. The big advantage of this scaling method is that, given the number of processors, the largest possible problem may be solved in every run.

5.2.2 Concurrency

Important new factors contributing to the execution time of parallel programs are communication and synchronization. Communication time is spent when data is sent from one processor to another one in the case of distributed-memory machines or when one processor accesses the global shared memory in the case of shared-memory machines. Synchronization is closely related to communication, since it occurs in general if one processor waits for another processor to update global data.

A very important issue for a parallel program is to keep the communication and synchronization times as small as possible, so that the speedup and the scaleup come close to ideal.

The *concurrency* of a parallel program is a broad measure for the number of processors that are, in some aggregate sense, simultaneously active in carrying out the computations of a given parallel program, that is they are not waiting for communication or synchronization. The degree of concurrency generally depends on the method by which the overall computation is split into smaller subtasks and is divided among the various processors for parallel execution.

For efficiency, it is important that the computation time of parallel subtasks is relatively uniform across the processors. Otherwise, some processors will be idle waiting for others to finish their subtasks. This aspect of parallel computing is known as *load balancing*. As the number of processors increases, the degree of concurrence and the amount of communication also increase. Both go hand in hand. Thus, by using more and more processors we must deal with an increased communication penalty. This may place an upper bound on the size of problems of a given type that we can realistically solve, even with an unlimited number of processors.

In any parallel or distributed algorithm, it is necessary to coordinate the activities of different processors to some extent. This coordination is often implemented by dividing the algorithm into *phases*. During one such phase each processor should operate independently from all the others, i.e., it should not synchronize and communicate. Communication and synchronization should mainly just take place in-between two phases. In one phase the program is said to run *asynchronously*. Keeping these phases as long as possible insures that the concurrency is maximal.

5.2.3 Synchronous Algorithms

A synchronous algorithm is an algorithm, where the start of each phase is simultaneous for all processors and the end of the message receptions is simultaneous for all messages. The implementation of a synchronous algorithm in an inherently asynchronous parallel machine environment requires a synchronization mechanism, that is, an algorithm that is superimposed to the original and by which each processor can detect the end of each phase. Such an algorithm

is called a *synchronizer*. There are two main approaches on which synchronizers are based: *global synchronization* and *local synchronization*.

In global synchronization all processors are synchronized at the end of each phase either by a build-in synchronizing barrier, by sending messages, or by setting semaphores. All processors start the next phase synchronously. In case of the message-passing model, the main idea is that if a processor knows which messages to expect in each phase, then it can start a new phase once it has received all those messages. This is known as *local synchronization*. The shared memory model does not have any implicit synchronization. However, two or more processors can be explicitly synchronized by using locks, for example. The communication penalty for local synchronization is considerably less than in the global synchronization case. There are factors, however, such as programming complexity, that favor the global method.

5.2.4 Asynchronous Algorithms

For each processor, there are times at which the processor executes some computation without any global data exchange and other times at which it exchanges data with other processors. The algorithm is said to be *asynchronous* if these times and thus also the order of computations and data exchange periods are not fixed *a priori* and can vary widely upon execution.

There are several potential advantages of asynchronous algorithms versus synchronous ones:

Reduction of Communication and Synchronization Overhead. Since data exchange is limited to just a few processors, only those processors have to be synchronized locally in case of the message-passing model. In case of the shared-memory model the only overhead is caused

by the cache-coherence as described in Section 5.6.3. The other processors that do not take part in the data exchange, operate independently.

Reduction of the Effects of Bottlenecks. Consider the following case: one or few processors are busy doing computations without updating their part of the global data and other processors are waiting for this new data in a local synchronization. In case of the asynchronous model processors, that do not depend on this data, continue their computation. Whereas in the synchronous model all processors have to wait for the data to be updated in a global synchronization. The asynchronous model facilitates a high concurrency. However, the local synchronization may cause a chain reaction in the asynchronous model, which can even cause *deadlocks*. That is, one processor is waiting for a second, where the second processor is waiting for the first. Both processors are waiting for each other and cannot continue their computation, forever. This is an infinite loop and causes the program to stall. Dead-locks can easily occur in rings caused by careless programming and using resources to their limits.

Convergence Acceleration due to Gauss-Seidel Effect. The parallel version of our algorithm has an effect like the serial *Gauss-Seidel iteration*, as discussed in the upcoming sections. It causes the solution to converge faster, since information is incorporated faster in the updated formulas. The newest information is used as soon it is available.

5.3 Jacobi Iteration

This and the subsequent section describe an iterative parallelized algorithm, which can be implemented in all SPMD programming models discussed in Section 5.1. Thus, this section and its successors are the foundation for all three versions of our program. First we describe the sequential form of this algorithm, the *Jacobi iteration*, to solve the linear algebraic equation (4.20) as given in Section 4.2:

$$\boldsymbol{x} = \tilde{\boldsymbol{A}} \cdot \boldsymbol{x} + \tilde{\boldsymbol{b}} \ .$$

The basic idea of the Jacobi iteration is to guess the solution $\boldsymbol{x}$ initially, then to insert this guess into the right-hand side of Equation (4.20). Evaluating the right-hand side gives a new and better approximation for the solution $\boldsymbol{x}$, which is inserted again into the right-hand side. This iteration step is repeated until the deviation of the approximation from the exact solution is sufficiently small.

Let $x_{(k-1)M+i}(t)$ denote the approximation of the solution φ_{ki} on the ith boundary element on the kth surface in iteration step t, then the initial guess is given by:

$$x_m(0) = b_m \ . \tag{5.5}$$

Inserting this initial guess into the right-hand side of (4.20) we get a new, more accurate guess $x_m(t+1)$:

$$x_m(t+1) = \sum_{n=1}^{NM} \tilde{A}_{mn} x_n(t) + b_m \qquad ; m = 1...NM \ . \tag{5.6}$$

The time $t \geq 0$ has dimensionless integer values and denotes the current iteration number.

The Jacobi iteration converges to the exact solution $\boldsymbol{x}$, if and only if the spectral radius of $\boldsymbol{A}$ is less than one, which is true for our problems, in general. The proof for the convergence is as follows. Let $\Delta\boldsymbol{x}(t)$ denote the absolute error of the approximation: $\Delta\boldsymbol{x}(t) = \boldsymbol{x}(t) - \boldsymbol{x}$. Using complete induction this error at iteration step t is given by $\Delta\boldsymbol{x}(t) = \tilde{\boldsymbol{A}}^t \cdot \Delta\boldsymbol{x}(0)$, which is a sequence converging to 0, if and only if the spectral radius of $\tilde{\boldsymbol{A}}$ is less than 1, assuming that the initial error $\Delta\boldsymbol{x}(0)$ is not equal to 0.

5.4 Parallelizing the Jacobi Iteration

In general, the parallelization of an algorithm involves two steps: First, distributing the data to the local address spaces of the different processors. Second, splitting up the computation, that is the work, to the different processors. It turns out that both steps melt into one step in the case of the Jacobi iteration. The solution vector for the double layer density $\boldsymbol{x}$ is split into subvectors, where each subvector describes the double layer density on all boundary elements on one body surface. In the same manner the system matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ are split. The parallel version of the Jacobi iteration features a small amount of global read and write data, which results in little communication between different processors, which is one of the reasons for its very good speedup and scaleup.

5.4.1 Splitting the Matrices

The solution vector $\boldsymbol{x}$ is split into N subvectors:

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_N \end{pmatrix} \tag{5.7}$$

where the subvector $\boldsymbol{y}_k$ describes the discretized double layer density φ_{ki} on all the boundary elements of the body k, that is $\boldsymbol{y}_k$ describes the complete solution on body k:

$$\boldsymbol{y}_k = \begin{pmatrix} y_{k1} \\ y_{k2} \\ \vdots \\ y_{kM} \end{pmatrix} = \begin{pmatrix} \varphi_{k1} \\ \varphi_{k2} \\ \vdots \\ \varphi_{kM} \end{pmatrix} = \begin{pmatrix} x_{(k-1)M+1} \\ x_{(k-1)M+2} \\ \vdots \\ x_{(k-1)M+M} \end{pmatrix} \quad ; k = 1...N \ . \tag{5.8}$$

In the internal program representation $\boldsymbol{x}$ is an array of 2nd order. Correspondingly $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{b}}$ are split into submatrices or subvectors respectively.

$$\tilde{\boldsymbol{A}} = \begin{pmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} & \cdots & \boldsymbol{C}_{1N} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} & \cdots & \boldsymbol{C}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{C}_{N1} & \boldsymbol{C}_{N2} & \cdots & \boldsymbol{C}_{NN} \ , \end{pmatrix} \tag{5.9}$$

$$\tilde{\boldsymbol{b}} = \begin{pmatrix} \boldsymbol{d}_1 \\ \boldsymbol{d}_2 \\ \vdots \\ \boldsymbol{d}_N \end{pmatrix} . \tag{5.10}$$

The submatrix $\boldsymbol{C}_{kl}$ represents the influence from body l onto body k. This interaction is independent of all other interactions in the system (the system is linear), so that this influence is expressed only in $\boldsymbol{C}_{kl}$, and in no other submatrix of $\tilde{\boldsymbol{A}}$, where $\tilde{\boldsymbol{A}}$ is an array of 4th order.

$$\boldsymbol{C}_{kl} = \begin{pmatrix} C_{kl11} & C_{kl12} & \cdots & C_{kl1M} \\ C_{kl21} & C_{kl22} & \cdots & C_{kl2M} \\ \vdots & \vdots & \ddots & \vdots \\ C_{klM1} & C_{klM2} & \cdots & C_{klMM} \end{pmatrix} \quad ; k = 1...N, \quad l = 1...N \ . \tag{5.11}$$

Each element of $\boldsymbol{C}_{kl}$ is calculated in the following way:

$$C_{klij} = \tilde{A}_{[(k-1)M+i][(l-1)M+j]} = A_{klij} \qquad ; k, l = 1...N, \quad i, j = 1...M \tag{5.12}$$

by using Equation (4.19) or Equation (4.22). Finally, $\boldsymbol{d}_k$ is a subvector of the whole constant vector $\tilde{\boldsymbol{b}}$ and $\tilde{\boldsymbol{b}}$ like $\boldsymbol{x}$ is an array of 2nd order. That is

$$\boldsymbol{d}_k = \begin{pmatrix} d_{k1} \\ d_{k2} \\ \vdots \\ d_{kM} \end{pmatrix} = \begin{pmatrix} \tilde{b}_{(k-1)M+1} \\ \tilde{b}_{(k-1)M+2} \\ \vdots \\ \tilde{b}_{(k-1)M+M} \end{pmatrix} = \begin{pmatrix} b_{k1} \\ b_{k2} \\ \vdots \\ b_{kM} \end{pmatrix} \qquad ; k = 1...N \ . \tag{5.13}$$

5.4.2 Splitting the Equations

This section describes the parallelized version of the Jacobi iteration formula (5.6). At first the synchronous version will be derived; the asynchronous version is presented in the next section. Each processor calculates $N_P = \frac{N}{P}$ subvectors, where P is the number of processors; that is, the bodies are uniformly distributed across the processors, assuming the number of bodies is a multiple of the number of processors. Every processor owns a subset of the iteration equations (5.6) as well as a subvector $\boldsymbol{d}_k$ of the constant vector $\tilde{\boldsymbol{b}}$, and is responsible for updating its set of solution subvectors y_k. Processor p has to evaluate the following formulas to update its subvectors within one iteration step:

$$y_{ki}(t+1) = \sum_{l=1}^{N} \sum_{j=1}^{M} C_{klij} y_{lj}(t) + d_{ki} \qquad ; k = [(p-1)N_P + 1]...(pN_P), \quad i = 1...M \ . \tag{5.14}$$

Written in tensor form:

$$\boldsymbol{y}_k(t+1) = \begin{pmatrix} \boldsymbol{C}_{k1} & \boldsymbol{C}_{k2} & \cdots & \boldsymbol{C}_{kN} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{y}_1(t) \\ \boldsymbol{y}_2(t) \\ \vdots \\ \boldsymbol{y}_N(t) \end{pmatrix} + \begin{pmatrix} \boldsymbol{d}_k \end{pmatrix} \tag{5.15}$$

$$k = [(p-1)N_P + 1]...(pN_P) \ .$$

This equation system is equivalent to the original Jacobi iteration (5.6). Figure 5.2 is a more

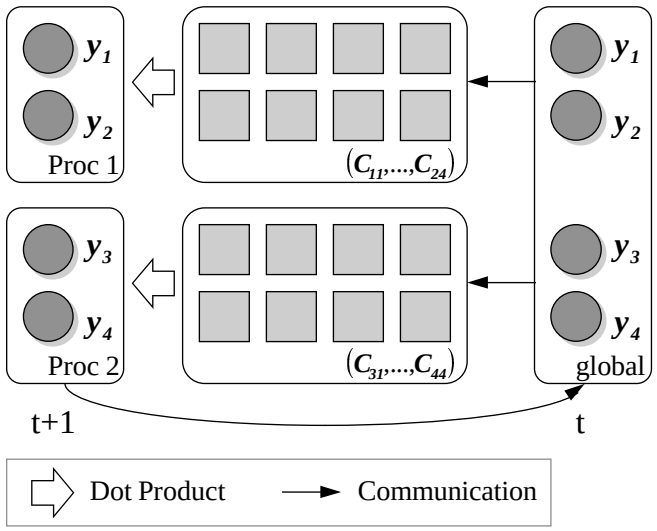

Figure 5.2: Parallel Jacobi iteration. Four bodies are distributed onto two processors. The solution on the bodies is communicated between all processors, which evaluate the dot products in parallel.

illustrative representation of this iteration scheme and shows the distribution of the bodies onto

the different processors. In order to calculate the solution on its bodies, processor p needs the following objects:

- the subvectors of the solution vector at timepoint (iteration) $t+1$ for the bodies on p:

$$\boldsymbol{y}_k(t+1) \qquad ; k = [(p-1)N_P + 1]...(pN_P) \ , \tag{5.16}$$

- the whole "old" solution vector at timepoint t:

$$\boldsymbol{y}_k(t) \qquad ; k = 1...N \ , \tag{5.17}$$

- all those submatrices of the system matrix, which describe the influence of all N bodies onto the N_P bodies belonging to processor p:

$$\boldsymbol{C}_{kl} \qquad ; k = [(p-1)N_P + 1]...(pN_P), \quad l = 1...N \ , \tag{5.18}$$

- N_P subvectors of the constant vector $\boldsymbol{d}$:

$$\boldsymbol{d}_k \qquad ; k = [(p-1)N_P + 1]...(pN_P) \ . \tag{5.19}$$

Except for the "old" solution vector at time t, all subvectors and submatrices can be stored in the local memory of processor p. The solution vector at time t either has to be stored in global memory in the shared-memory versions or has to be communicated by messages in the

message-passing version.

The iteration scheme (5.15) is a *synchronous algorithm* as described in Section 5.2.3. One iteration step (5.15) includes a dot product, an addition, and an assignment, which can be executed independently in a parallel phase, i.e., asynchronously. But at the end of this iteration step there is a synchronization barrier, at which the solution subvectors are communicated between all participating processors. Each processor has to know the solution at time $t+1$ of all processors in order to execute the next iteration step at time $t+2$. This leads to synchronization overhead, since some processors may take longer to execute the iteration step than others, especially when parts of the system matrix $\boldsymbol{C}$ have to be recalculated, as discussed in Section 5.4.4. The faster processors have to wait for the slower processors and are in the idle state. A large amount of communication overhead also occurs in both the message-passing and the CSM versions: at the end of each iteration step all processors distribute all their solution vectors at the same time, where the network or the shared memory may become a bottleneck. To get around these problems, this iteration scheme is modified, so that there is no synchronization necessary at the end of each iteration step.

5.4.3 Asynchronous Iteration

A different and better approach is an *asynchronous iteration* scheme where a solution subvector on a distant body is communicated, as soon as and only when it is needed. This guarantees that the most recent solution vector is always used, which accelerates the convergence of the solution as the Gauss-Seidel Iteration does in the sequential case [5]. Furthermore, the synchronization

and communication overhead is reduced.

Removing the synchronization barrier gives us more freedom in choosing a suitable iteration scheme. The physical insight into the problem of charged bodies tells us that the local interactions between bodies are dominant against the interactions between bodies that are far apart. Thus, we do not want to consider the influence from a body that is far apart onto a particular body as often as we consider the influence from a close body in our calculations. To update an influence from one body k onto another body l means to communicate the solution on body k to body l. This idea improves the performance of the algorithm implementation in two ways:

1. If the two considered bodies are calculated on two different processors, the amount of communication is reduced, since the solution on the bodies are exchanged less frequently between the two processors. This demands that neighboring bodies should be clustered on one processor, so that the computation of the solution on neighboring bodies can be executed locally on the processors without any communication.

2. The amount of computation is reduced, since a processor only evaluates the dot product associated with a solution subvector, if this solution subvector has been updated in its local address space. Furthermore, if the submatrix $\boldsymbol{C}_{kl}$ needed for this dot product is not stored in memory but has to be recreated because of memory shortage, the decrease in computation is of orders of magnitude.

At the present time, the problem size is limited by the computational performance of current parallel machines, and item 2 dominates. However, on future machines item 1 may become

more important as memory and network performance lag behind microprocessor development.

The implementation of this modified iteration scheme is as follows. A further input to the algorithm is the *communication schedule matrix (CS matrix)* $\boldsymbol{S}$, whose integer elements describe the degree of influence from one body onto another. This $N \times N$ matrix is constructed *a priori*. The element s_{kl} minus 1 is the number of iterations, in which the update of the influence from body l onto body k is suppressed. That is, the influence is only updated in those iteration steps t, where the division of t by s_{kl} yields no remainder. In those iteration steps the processor calculating body k requests the solution on body l from its processor or accesses global shared memory to obtain the solution on body l. After this it recreates the corresponding submatrix $\boldsymbol{C}_{kl}$ if necessary and updates the corresponding dot product.

Note that there is no global synchronization occuring in this modified iteration scheme. However, there is local synchronization occuring in the message-passing and the Split-C versions implemented by a *request-and-send scheme*. For instance, the implementation in the Split-C version is based on global assignments (see Section 5.6.2), which the Split-C compiler translates into request-and-send messages. The CSM version does not have any local synchronization.

A mathematical representation of this iteration scheme is shown in the subsequent paragraphs for one processor p, where $1 \leq p \leq P$. Processor p updates the solutions on its N_P

bodies in one iteration step as follows:

$$\boldsymbol{y}_k(t) = \begin{pmatrix} \boldsymbol{C}_{k1} & \boldsymbol{C}_{k2} & \cdots & \boldsymbol{C}_{kN} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{y}_1(t_{k1}) \\ \boldsymbol{y}_2(t_{k2}) \\ \vdots \\ \boldsymbol{y}_N(t_{kN}) \end{pmatrix} + \begin{pmatrix} \boldsymbol{d}_k \end{pmatrix} \tag{5.20}$$

$$k = [(p-1)N_P + 1]...(pN_P), \quad \max_l\{t_{kl}\} \leq t < \min_l\{t_{kl} + s_{kl}\} \ .$$

Here t_{kl} denotes the latest iteration step, in which the solution subvector $\boldsymbol{y}_l$ on body l has been updated on processor p. That is, the last time processor p has evaluated the dot product $\boldsymbol{C}_{kl} \cdot y_l$. Note that this scheme still converges to the correct solution because the eigenvalues of the system matrix $\boldsymbol{C}_{kl}$ are dominated by the eigenvalues of its diagonal submatrices, which describe the influence from one body onto itself (see Section 7.2). The execution of the iteration may be divided into distinct phases, in which the solution converges to some local fixed point on one processor, but this fixed point moves at that time, when the influence from a distant body is updated. This starts a new phase and a new fixed point is approached, until another long range interaction is updated.

The scheme shown in Equation (5.20) is not optimal, since the dot products have to be evaluated in each iteration step, even when the corresponding solution subvector has not been updated. A modified improved scheme does not calculate the new solution in the current iteration step from scratch, but calculates the difference between the new solution $\boldsymbol{y}_k(t)$ and the

old solution $\boldsymbol{y}_k(t-1)$. This approach has the advantage that not all of the dot products have to be evaluated, but only those, which correspond to the updated solution subvectors $\boldsymbol{y}_l(t)$. The drawback is that the old dot products have to be stored in the array $\boldsymbol{z}_{kl}(t_{kl}) = \boldsymbol{C}_{kl} \cdot \boldsymbol{y}_l(t_{kl})$. But the penalty of storing these dot products is outweighed by the significant decrease in computation time. This is the mathematical representation of the improved iteration scheme:

$$\boldsymbol{z}_{kl}(t_{kl}) = \boldsymbol{C}_{kl} \cdot \boldsymbol{y}_l(t_{kl})$$

$$\boldsymbol{y}_k(t) = \boldsymbol{y}_k(t-1) + \sum_{l=1, t_{kl}=t}^{N} [\boldsymbol{z}_{kl}(t_{kl}) - \boldsymbol{z}_{kl}(t_{kl} - s_{kl})] \tag{5.21}$$

$$k = [(p-1)N_P + 1]...(pN_P), \quad \max_l\{t_{kl}\} \leq t < \min_l\{t_{kl} + s_{kl}\} \ .$$

The element of the CS matrix s_{kl} is interpreted as a priority, which denotes the degree of the influence from body l onto body k. The highest priority is $s_{kl} = 1$. In this case the corresponding solution subvector $\boldsymbol{y}_l$ is communicated to processor p in each iteration step. A straightforward approach for setting the elements s_{kl} is taking a feature that is closely related to the distance between the bodies k and l. In our approach s_{kl} is proportional to the distance between the bodies k and l, scaled by the maximum distance occuring in the given array of bodies. There might be better criteria for the degree of influence between two bodies, e.g. as presented in [17], but our choice already speeds up the execution time significantly.

5.4.4 Creating the System Matrices

This section describes how the system matrix $\boldsymbol{C}$ and the constant vector $\boldsymbol{d}$ are calculated. The vector $\boldsymbol{d}$ is calculated completely before the iteration algorithm is started, whereas the system matrix $\boldsymbol{C}$, which uses a huge amount of memory, is optionally calculated only partially to save memory. Of course, we want to use the whole main memory on the computer, so we store as many submatrices as possible in memory. Those submatrices stay in memory for the whole iteration. Whenever the non-stored submatrix $\boldsymbol{C}_{kl}$ is needed in the iteration scheme (5.21) it has to be calculated in this iteration step. After the corresponding dot product $\boldsymbol{C}_{kl} \cdot \boldsymbol{y}_l$ has been evaluated, the memory for $\boldsymbol{C}_{kl}$ is relinquished and can be used for the storage of a different submatrix.

One submatrix $\boldsymbol{C}_{kl}$, which describes the influence from body l onto body k, has M^2 elements. For each element we have to calculate the integral of the kernel K over the boundary element area of body l, as described in Equation (4.19) and Equation (4.22). If we assume the distance between the two involved boundary elements to be large, then each calculation of a submatrix $\boldsymbol{C}_{kl}$ causes an execution of a double loop. If this distance is small, then an even more expensive quadruple loop is to be executed, because Q-point Gaussian quadrature (with $Q > 1$) is used to integrate the kernel.

Because of the high computational expense of calculating a submatrix $\boldsymbol{C}_{kl}$, frequently used submatrices are calculated before the iteration and stored in memory permanently, whereas submatrices that describe long range interactions are not stored permanently but are calculated when they are needed. As a criterion for the priority of a submatrix $\boldsymbol{C}_{kl}$ we use the values of the

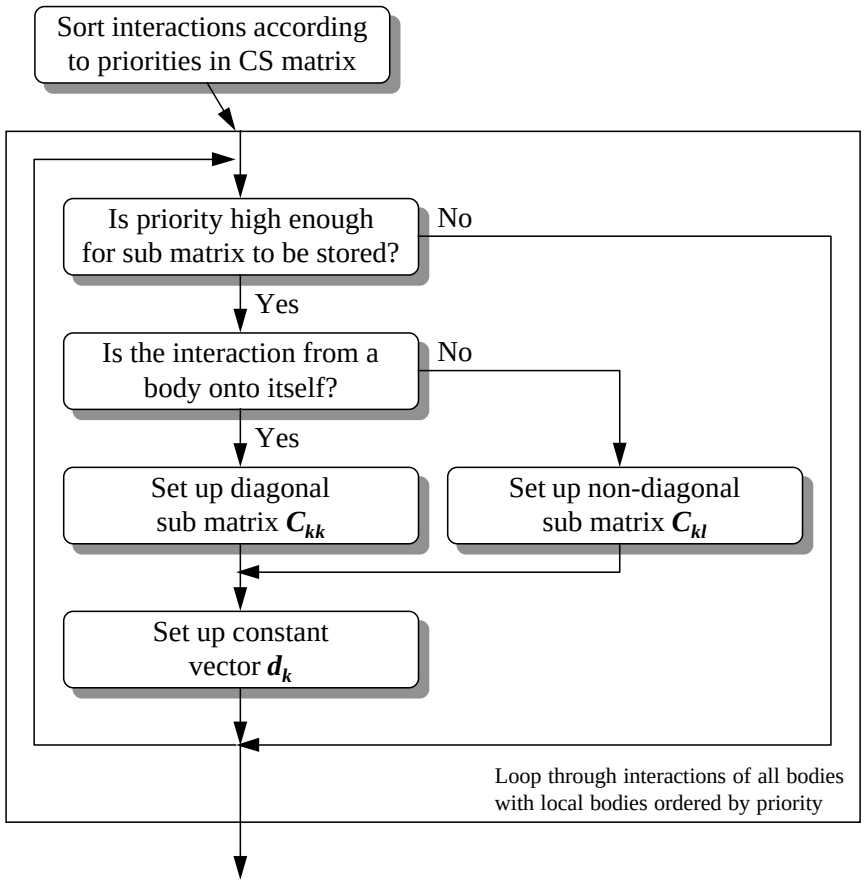

Figure 5.3: Matrix set-up before the iteration. The routines for setting-up the diagonal matrix $\boldsymbol{C}_{kk}$ and the non-diagonal matrix $\boldsymbol{C}_{kl}$ are also called during the iteration.

elements in the CS matrix $\boldsymbol{S}$. The submatrices are sorted by their priority and the submatrices are stored permanently in memory according to this order. This ensures that submatrices with high priorities, e.g. the submatrices which describe the interactions from a body onto itself, are always kept in memory, and that submatrices with low priorities are not stored but calculated during the iteration. Figure 5.3 shows the routine that controls the setup of the submatrices before the iteration.

5.5 Load Balancing

The load balance of our program has a large impact on its scaling. In order to optimize its scaling, the bodies are distributed onto the processors in such a way, that each processor has approximately the same workload which guarantees an optimal load balance. We use the CS matrix to estimate the amount of computation time needed to calculate the solution on one body. This estimation is the sum of all entries in row k of the CS matrix $\boldsymbol{S}$ which corresponds to body k:

$$g_k = \sum_{l=1}^{N} s_{kl} \ . \tag{5.22}$$

The greater g_k is, the less often influences of other bodies onto body k are updated, and the less communication and computation is executed. The sum of these criteria g_k over all bodies on one processor p

$$g^p = \sum_{k=(p-1)N_P+1}^{pN_P} g_k \tag{5.23}$$

should have a minimal variance with respect to the processors. This problem is called *off-line makespan scheduling* [9], where N nonnegative numbers, g_k, are partitioned into P blocks. The sum of the numbers in one block p, g^p, have a maximum. This maximal sum has to be minimized, so that the variance of g^p is minimal. A further constraint is, that each of the P processors has to have the same number of bodies N_P. This problem is $\mathcal{NP}$-complete. It is generally believed that algorithms solving $\mathcal{NP}$-complete problems both exactly and efficiently do not exist. For this reason, a heuristic algorithm is used to solve this problem. We use a reversed and modified form of *Largest Processing Time first (LPT)* [21]. Our algorithm is *reversed* because we take the "smallest" processing time first, and *modified*, since each processor has to have the same number of bodies.

The load balancing routine works as follows:

1. The bodies are sorted so that g_k is in a strictly decreasing order.

2. Body k with the greatest g_k is assigned to the processor with the least g^p, and g^p is updated.

3. Then body k_2 with the second greatest g_{k_2} is assigned to the processor which now has the least g^p.

4. The previous step is iterated until all bodies are distributed.

This routine guarantees that the variance of g^p is minimal and optimizes the load balance of the processors.

However, LPT is not optimal as illustrated in the following example. Consider a set of $N = 5$ numbers

$$\{g_k\} = \{5, 4, 3, 3, 3\} \ .$$

These numbers are partitioned into $P = 2$ blocks. LPT yields the following two sets for the two blocks:

$$\{5, 3\} \qquad , \qquad \{4, 3, 3\} \ .$$

But the optimal solution is

$$\{5, 4\} \qquad , \qquad \{3, 3, 3\} \ .$$

However, if N is much larger than P, both the absolute and the relative error of LPT tend to zero [9].

5.6 Implementation in Parallel Programming Models

The implementation of the algorithm in a programming model is used to analyze the performance both of the algorithm and of the programming model. This analysis indicates whether the algorithm can be applied to large-scale problems and which programming model is most promising.

There are three implementations of the algorithm in different programming models: one in ANSI-C using the message-passing library CMMD 3.0 on the CM-5, one in Split-C 1.0 and one in CSM. Differing from the first two implementations the CSM implementation does not assume

the CM-5 as the underlying hardware. Its underlying hardware is Dir_1SW, which is discussed in [24]. This section discusses the different aspects of these implementations and presents the used programming models in more detail.

Common to all implementations is that the program consists of the five parts:

Initialization	read input files
	initialize timers
	distribute data to each processor
	initialize data
Discretization	discretize boundaries into boundary elements
Equation Set-up	calculate system matrices
Iteration	calculate double layer density by iteration
Output	collect data and write output files

Depending on the programming environment, the initialization and the output are done either in parallel or on one processor that distributes the input data to the other processors and collects the output data from them. All implementations execute the discretization of the boundaries in parallel, that is, all processors do the same job. A more elegant way would be to parallelize this job as well and let the processors exchange their data after it. However, since the discretization is very fast compared to the other parts, the introduced overhead is negligible. After having done the discretization, each processor continues without any synchronization, which would have to be inserted in the case of a parallel implementation of the discretization, and sets up its own part of the linear equation system without any synchronization or communication. In between this part and the iteration is a synchronization barrier to ensure that each processor has set up its equations before the asynchronous iteration routine starts. Thus, the iteration is started simultaneously and is concluded by a further synchronization. After that

the results are collected and written to file either from one processor or from all processors in parallel, depending on the programming environment.

Since most of the differences of the implementations are in the iteration routine and the features of the programming models are mostly expressed in this routine, the following sections only discuss the implementation of the iteration routine in more detail. Figure 5.4 shows the phases of the iteration algorithm.

5.6.1 ANSI-C and CMMD

This programming model is the standard message-passing model on the CM-5 as suggested by Thinking Machines Corporation [49]. To understand this model a brief introduction into the CM-5 hardware is given [48].

The CM-5 is a distributed-memory, message-passing parallel machine. It has comparatively powerful workstation-like processing nodes that include a 22 MIPS SPARC processor, a network controller, up to 32 MByte of DRAM and optional four 32 MFLOPS vector units. The SPARC processor has a 64 KByte instruction and data cache. All the processing nodes are connected by two different networks accessible to user programs. The *data network* is used to communicate data between the nodes and its topology is a fat tree. That is, four nodes are grouped together by a network switch and four communication channels. Four of these groups are connected together by another switch and communication channels with a higher bandwidth as shown in Figure 5.5. The other network, the *control network*, is used to coordinate the processing nodes and has different purposes for different programming models. For instance, the CM-5 can be

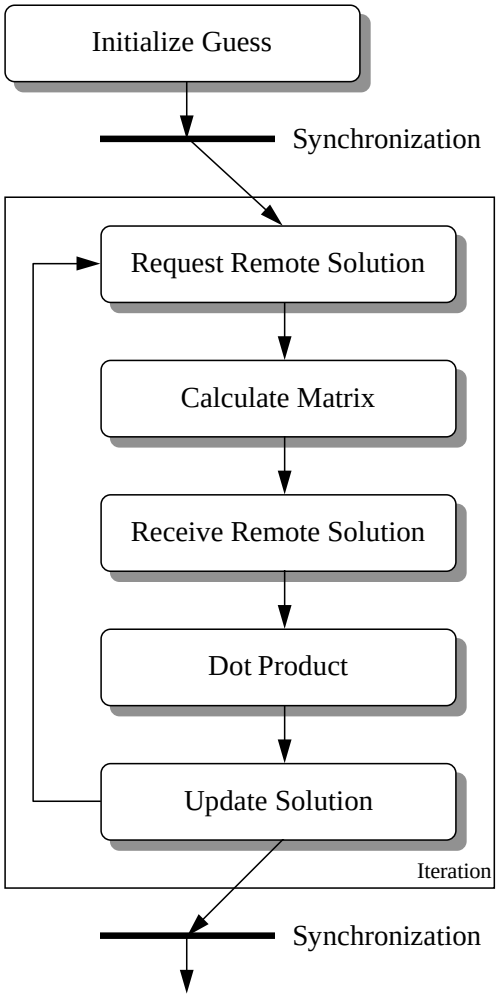

Figure 5.4: Iteration algorithm. The CSM implementation does not have the block *Request Remote Solution*.

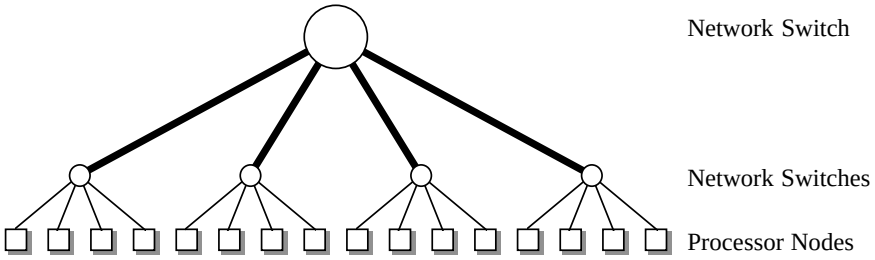

Figure 5.5: Data network of the CM-5.

used as a SIMD machine to run the data-parallel model, where the control network provides global synchronization and global data operations, such as summing and broadcasting, on a hardware basis.

The standard way to use the CM-5 as a MIMD machine is to write the program code in C, C++, CM-Fortran or C* and call the library functions of the CMMD library. Because of history (the former CMMD version 2.0 did not support the data-parallel languages CM-FORTRAN or C*) our program is coded in C. In addition to message-passing routines, the CMMD library supports functions for parallel disk I/O, timers, broadcast and scan functions, and synchronization barriers. Upon execution the program is distributed to all processing nodes of a CM-5 partition, which run the program in parallel. This seems to contradict the MIMD model that claims that each processor should run a different instruction sequence. However, by calling the CMMD function `CMMD_self_address` each processor obtains a unique number from 0 to $P-1$, which it uses to branch to its own dynamic instruction sequence.

The data in both implementations, in CMMD and Split-C, are distributed in the same

`N`	number of bodies N
`NP`	number of bodies on one processor N_P
`C[kk][l]`	system submatrix $\boldsymbol{C}_{kl}$ – processor stores only its submatrices
`P`	number of processors P
`S[kk][l]`	CS matrix entry s_{kl} – processor stores only its entries
`d[kk]`	constantsub vector $\boldsymbol{d}_k$ of the equation system – processor only stores its own subvectors
`flag`	boolean flag
`iter`	iteration number t
`kk`	loop variable – loops through bodies on the local processor
`l`	loop variable – loops through all bodies
`ll`	loop variable – loops through bodies on the remote processor `p`
`me`	number of the local processor
`p`	loop variable – processor number
`y[kk]`	solution subvector $\boldsymbol{y}_k(t+1)$ – processor stores only its own subvectors
`yold[k]`	solution subvector $\boldsymbol{y}_k(t)$ – each processor stores whole vector locally
`z[kk][l]`	dot product $\boldsymbol{z}_{kl}$ – processor stores only its dot products

Table 5.1: Objects in the C+CMMD and Split-C codes.

way. The matrix $\boldsymbol{C}$, the constant vector $\boldsymbol{d}$ and the solution vector $\boldsymbol{y}(t+1)$ at time $t+1$ are distributed as described in Section 5.4.2. Each processor has a copy of the whole solution vector $\boldsymbol{y}(t)$ at time t, which basically is the input buffer for solution subvectors received from distant processors. Each processor also has a copy of the boundary elements' data to allow a fast calculation of the system matrices without any communication. The object names and their corresponding mathematical notations as introduced in Section 5.4.3 are summarized in Table 5.1.

The iteration routine is shown in a high abstraction level in Figure 5.8. It uses the two

```
/* initialization */
Loop kk=1..NP  // loop through local bodies (rows)
  yold[me*NP+kk] = d[kk]  // initialize y_k with first guess d_k
  Loop l=1.. N  // loop through all bodies (columns)
    z[kk][l] = 0.  // initialize dot products with 0
  End Loop l
End Loop kk
```

Figure 5.6: Subroutine INITSOLUTION() – initialization (first guess) of the solution.

```
/* message-loop for requests */
Loop While (REQUEST?(p) == True)  // loop while there are requests
  /* REQUEST returns requesting processor in p */
  SEND(yold[me*NP+1..(me+1)*NP] to p)  // send local solution to
                                       // processor p
End Loop
```

Figure 5.7: Subroutine MSGLOOP() – message-loop for requests.

routines `INITSOLUTION()`, which is common to all programming models, and `MSGLOOP()`, as described in Figure 5.6 and Figure 5.7. After setting the solution vector $\boldsymbol{y}$ to the initial guess in `INITSOLUTION()`, a barrier is executed synchronizing all processors to guarantee that the initialization has been executed on all processors. Then the iteration is started.

Our implementation of the communication is based on a *request-and-send scheme*, where processor A sends out a request message to processor B, when it needs the solution subvectors from processor B for its calculations. As soon as processor B has received this request, it sends its solution to processor A. This request-and-send scheme is implemented in such a way that communication overlaps computation and the processors have to wait as little as possible for messages. Each processor executes a so called *message-loop*[4] at least once in an iteration step, where it looks for incoming requests, which are buffered in an operating system operated buffer

[4]This term is stolen from IBM OS/2 or MS Windows.

```
INITSOLUTION()  // initialize solution with 0
BARRIER()  // synchronize all processors
/* iteration */
Loop iter = 0..(ITER_MAX-1)
  /* request solution from remote processors */
  Loop p = 1..P  // scan remote processors
    flag = False
    Loop ll = 1..NP  // scan bodies on remote processor p
      Loop kk = 1..NP  // scan local bodies
        If (iter % S[kk][(p-1)*NP+ll] == 0)  // check if to update influence
          flag = True  // memorize request
        End If
      End Loop kk
    End Loop ll
    If (flag == True && p != me)
      SEND(Request to p for solution on all its bodies)  // send request
    End If
  End Loop p
  /* update local solution */
  Loop kk = 1..NP  // loop through local bodies (rows)
    Loop l = 1..N  // loop through all bodies (columns)
      If (iter % S[kk][l] == 0)  // check if to update influence
        /* influence from remote body l onto local body kk must be
           updated */
        If (C[kk][l] is not stored in memory)
          /* sub matrix C_kl is calculated */
          CREATE(C[kk][l])
        End If
        /* wait for message containing requested solution on body l */
        Loop
          MSGLOOP()  // message-loop for requests
        End Loop Until (RECEIVED?(yold[l]) == True)
        /* dot product */
        dot = C[kk][l] . yold[l]
        y[kk] += dot - z[kk][l]  // update by difference
        z[kk][l] = dot  // store dot product for next update
      End If
    End Loop l
    yold[me*NP+kk] = y[kk]  // switch times t <-> t+1
  End Loop kk
End Loop iter
/* synchronize all processors */
Loop While (FINISHED?() == False)  // wait until all processors are done
  MSGLOOP()  // message-loop for requests
End Loop
```

Figure 5.8: Iteration algorithm in C+CMMD.

for incoming messages from the network. Only if there are no requests, the processor continues with its computation, otherwise it responds to the requests. The usage of the *asynchronous point-to-point messages* of CMMD avoids deadlocks.

In the first phase of the iteration, the processor scans the CS matrix $\boldsymbol{S}$ for interactions between the particles that it has to update in the current iteration step. It sends out messages requesting the corresponding solution subvectors on distant processors, and continues creating the corresponding submatrices if they are not stored in memory. In this phase communication and computation are overlapped. After a submatrix is reconstructed, the processor has to wait for the corresponding solution subvector to come in so that it can evaluate the dot product. During this wait the processor is not idle, but executes the message-loop and responds to requests from other processors. This nesting of waiting-for data and waiting-for requests guarantees that there are no deadlocks occuring. After the processor has evaluated all its dot products, it updates its part of the solution vector and starts the next iteration step. The number of iteration steps is given *a priori* in the input data. When the iteration is done on the fastest processor, care must be taken since other processors are still requesting data from this processor. It cannot just stop but has to execute a further message-loop in order to respond to those requests. When all processors have finished their iteration, their results are collected using the CMMD broadcast and scan functions and the output files are created.

We use the request-and-send scheme, since the requests for remote solution subvectors are infrequent. In a different approach, which would not have the penalty of sending requests, processor B would decide by itself, if it has to send its solution subvectors to processor A. The

request-and-send scheme does not have the overhead of making this decision twice, once on the receiving processor A and once on the sending processor B. By sending the requests at the beginning of the iteration step and by overlapping the request messages with computation, the penalty of sending the requests is kept low. Furthermore, the request-and-send scheme includes less local synchronization of processor A and processor B.

5.6.2 Split-C

Split-C supports SPMD programming by adding parallel computing directives to ANSI-C, such as global data assignments, barriers, atomic functions, and locks [11, 12]. Split-C's programming model is the static shared memory (or non-uniform memory access, NUMA) model where each processor stores a fixed portion of global memory and communicates with other processors in order to access the other portions of global memory. Split-C's address space is 2-dimensional where one coordinate is the processor number and the other one the local address.

The underlying hardware of Split-C is also the CM-5. It uses the data network of the CM-5 very efficiently by basing the communication on *active messages* [13]. Active messages are asynchronous point-to-point messages that have a built-in handler, which is called upon arrival of the message on the target processor B. As shown in Figure 5.9, the sending processor A sends the active message via the data network to processor B that is interrupted by this handler. Processor B plays a passive part in receiving this message: it just executes the handler, which is protected from the current execution state of processor B. One task of this handler is to write data into the address space of processor B, so that processor B can access this data upon

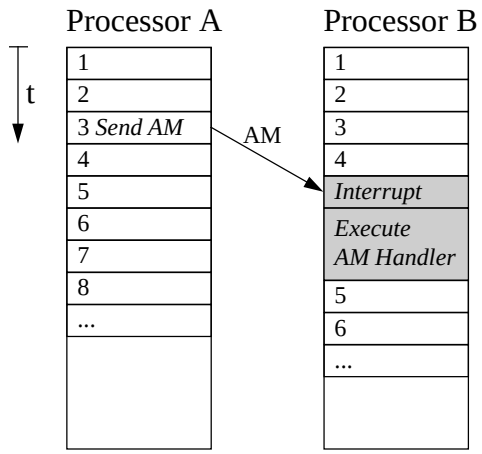

Figure 5.9: Active message. Processor A executes its instruction set and issues a send of an active message AM. Processor B receives this message and its instruction set is interrupted. During this interrupt the AM handler is executed. Upon return of the interrupt, processor B resumes its state at the start of the interrupt and continues the execution of its instruction set.

return of the handler. Since Split-C uses the SPMD model, the addresses of the handler both on processor A and processor B are the same, and the handler function is existent in the code segments of both processors. Note that the messages in version 3.0 of CMMD are also based on active messages.

Split-C provides the following extensions to ANSI-C:

Global Pointers. Global pointers are an extension to C pointers. They have a 2-dimensional address: the processor number and the local address.

Spread Arrays. Spread arrays are arrays, which are distributed over all nodes. The associated pointers to spread arrays are *spread pointers.*

Split-phase Assignment. Using the original C assignment (`=`) with a global pointer accesses memory on a distant processor, where the accessing processor has to wait until the communication is done. Using split-phase assignments (`:=`) a requesting processor sends out a request to a distant processor and continues with the execution of its instruction set until the next synchronization. *Stores* (`:-`) operate in the same way, but have a weaker synchronization.

Atomic Functions. Split-C provides functions, which run atomically and cannot be interrupted during execution.

Locks. Locks protect the consistency of variables.

Barriers. Barriers synchronize the processors.

The data distribution in the Split-C implementation is the same as in the CMMD implementation (see Section 5.6.1 and Table 5.1).

In the Split-C program (see Figure 5.10), the communication of the solution subvectors is also based on a request-and-send scheme, similar to the one in the message-passing program. However, in Split-C no message-loop has to be implemented as in the CMMD version, since the request-and-send scheme is provided internally. Split-C translates each split-phase assignment into a sequence of active messages. Thus, instead of sending requests at the beginning of an iteration step, a processor issues a split-phase assignment to get the solution subvector from a distinct processor, if it needs it. Since the split-phase assignment overlaps the computation, the processor continues the execution of its instruction set. At the point where it starts evaluating

```
INITSOLUTION()  // initialize solution with 0
BARRIER()  // synchronize all processors
/* iteration */
Loop iter = 0..(ITER_MAX-1)
  /* request solution from remote processors */
  Loop p = 1..P  // scan remote processors
    flag = False
    Loop ll = 1..NP  // scan bodies on remote processor p
      Loop kk = 1..NP  // scan local bodies
        If (iter % S[kk][(p-1)*NP+ll] == 0)  // check if to update influence
          flag = True  // memorize request
        End If
      End Loop kk
    End Loop ll
    If (flag == True && p != me)
      /* issue split-phase assignment to get solution from processor p */
      GET(yold[(p-1)*NP+1] .. yold[p*NP])
    End If
  End Loop p
  /* update local solution */
  Loop kk = 1..NP  // loop through local bodies (rows)
    Loop l = 1..N  // loop through all bodies (columns)
      If (iter % S[kk][l] == 0)  // check if to update influence
        /* influence from remote body l onto local body kk must be
           updated */
        If (C[kk][l] is not stored in memory)
          /* sub matrix C_kl is calculated */
          CREATE(C[kk][l])
        End If
        /* synchronize split-phase assignment for solution on body l */
        SYNC_GET(yold[l])
        /* dot product */
        dot = C[kk][l] . yold[l]
        y[kk] += dot - z[kk][l]  // update by difference
        z[kk][l] = dot  // store dot product for next update
      End If
    End Loop l
    yold[me*NP+kk] = y[kk]  // switch times t <-> t+1
  End Loop kk
End Loop iter
BARRIER() // synchronize all processors
```

Figure 5.10: Iteration algorithm in Split-C.

the dot product, it synchronizes the split-phase assignment, that is, it waits for the assignment to complete.

Although Split-C's programming model is the static shared-memory model, the program has two message-passing related features: synchronization of the split-phase assignments and additional message-polling function calls: `CMAM_poll`. In Version 1.0 of Split-C the receiving processor only checks for incoming active messages at split-phase assignments or other Split-C specific directives. This contradicts to the definition of active messages, which interrupt the receiving processor. The checking for incoming messages is called *polling* and is much too infrequent in our program, since split-phase assignments occur very infrequently and the delay in receiving active messages puts the sending processor into idle state, which slows down the execution or even can cause deadlocks. By inserting additional function calls `CMAM_poll` to the active messages library at appropriate locations causes the processor to poll for messages more frequently and decreases the delay in processing messages.

Split-C is a very promising language in a static shared-memory model. Our program was easier to code and is more transparent than the implementation in the message-passing model using the CMMD library. In addition, the performance of the Split-C communication is better than the message-passing of the CMMD library and reduces the communication overhead in our program significantly as shown in Section 7.3. However, the principal structure of the program is the same and as much care must be taken in implementing an efficient communication scheme, since the communication in the current Version 1.0 of Split-C is closely related to the message-passing model.

5.6.3 Cooperative Shared Memory

The first part of this section gives a brief introduction into the *Cooperative Shared Memory CSM* model implemented as the *CICO model* as described in more detail in [24]. In the second part, special issues of the algorithm's implementation in this model and the key differences to the message-passing and Split-C implementations are discussed. This section is also covered in our paper [51].

A widespread belief is that shared memory and shared-memory hardware is not scalable because of the assumption that all memory references have the same cost. This assumption is incorrect, since access to remote memory requires communication and, in orders of magnitude, is slower than the access to local memory. Furthermore, existing shared-memory hardware, whose implementations so far are small-scale shared-memory architectures based on bus systems, does not scale to highly parallel systems, or does so only in connection with a large increase of complexity because of their limits in bus capacity and bandwidth.

An alternative approach is a hardware with directory-based cache-coherence protocols leading to *cache-coherent parallel computers*. In this architecture each processor has local cache and is connected to a data network in such a way that the programmer and the compiler see a uniform address space. Note that this model is different to the NUMA model of Split-C, since it is a dynamic shared-memory model where local copies of shared data are automatically updated and the programmer does not have to know the physical location of the shared data. The advantages of shared memory are obvious, including a uniform address space, which allows the construction of distributed data structures, and referential transparency, which provides object

names (addresses) that are identical for both local and remote objects. Distributed data structures facilitate fine-grained sharing, free programmers and compilers from per-node resource limits. The uniform semantics simplify programming, compilation, and load balancing.

We implemented our algorithm in a special form of the shared-memory model, the so-called *cooperative shared memory CSM* model. This model is cooperative in two ways. First, it discourages programs whose processers compete when accessing data and avoids unnecessary communication. Second, it identifies sharing patterns that the hardware can support effectively to encourage their use, thus, allowing the cooperation between software and hardware.

A key component of CSM is its implementation of a performance model which provides insight into the program's behavior for both programmers and computer architects. This programming model is called the *Check-in / Check-out (CICO) model*, which is a design tool for programs on a cache-coherent parallel computer. It provides *check-in* and *check-out* annotations that bracket uses of shared data. Check-out indicates that a processor expects to use a shared object, and check-in terminates an expected use. This performance model includes two functionalities. First, the descriptive functionality allows the program to indicate where it expects communication during its execution. Second, the prescriptive functionality provides a cost model which decribes how the programmer can reduce the cost of this communication.

CICO's annotations are:

`check_out_X`	Expect exclusive access to data
`check_out_S`	Expect shared access to data
`check_in`	Expect end of data access

`check_out_X` asserts that the processor performing a check-out expects to be the only pro-

cessor accessing the block until it is checked in. `check_out_S` asserts that the processor is willing to share (read) access to the block. Exclusive access is not synonymous with write access. When a processor wants to read a block and expects to be the only reader, it should use `check_out_X` because `check_out_S` is more expensive and more difficult to manage in the hardware. `check_in` asserts that the access is done. From the programmer's point of view, check-out may be viewed as fetching a copy of the data into the processor's cache, and check-in as flushing the data from the cache into shared memory.

When a CICO program executes on a cache-coherent computer, communication will occur for three reasons. First, communication occurs at `check_out` and `check_in` annotations. Consequently, performance can be improved by restructing the code to move the annotations out of inner loops whenever possible. Second, communication occurs when the program violates an annotation's assumption (for example, the programmer expects exclusive access and multiple processes access the same data). Third, communication takes place whenever there is *false sharing*. Caches are organized in *cache blocks*. One cache block contains a fixed amount of words and is aligned to an address, which is a multiple of the cache block size. If an object is referenced in the main memory or in global shared memory, not only this one object but also its surrounding objects, according to the boundaries of the cache block, are transferred to the cache. When fetching a datum in a cache block, the whole cache block with more than one datum is fetched. If other processes require exclusive access to other data in this cache block, false sharing occurs.

In the future, CICO annotations will be associated with high-level language data types, so

that the compiler can manage both large data aggregates and eliminate false sharing – up to now the programmer has to take care of that. The key advantage of the CICO model is that its functionality and its performance are orthogonal, whereas in message-passing, communication is explicitly and inextricably linked with functionality. Evaluating the cost of communication is difficult since it must be evaluated twice: once on the sending processor and once on the receiving processor. Improving a message-passing program is a difficult task, since a small change can cause a cascade of modifications and has to be done carefully to avoid transformations that could introduce functional bugs.

Using caches a programmer has to consider several rules for a fast program execution. Since caches are arranged in cache blocks, a key issue for the program's performance is the *locality* of its data. There are two different forms of locality [23]. *Spatial locality* occurs because two simultaneous memory references are likely to access nearby words. *Temporal locality* arises because a recently referenced memory word is likely to be accessed again. Therefore, the data in our program is organized in blocks according to these locality forms. For instance, one memory block contains all the data of all boundary elements of one body because this data is likely to be accessed within a short time period during the calculation of one submatrix of $\boldsymbol{C}$. This organization of the memory is used for all data. To avoid false sharing, data aggregates are aligned according to the alignments of the cache blocks, so that the first word in a cache block is the first word in the data aggregate. The most expensive form of false sharing, where one cache block contains both read-only and writable data, is also avoided by distributing read-only and writable data aggregates into different memory blocks.

`N`	number of bodies N
`NP`	number of bodies on one processor N_P
`C[kk][l]`	system submatrix $\boldsymbol{C}_{kl}$ – processor stores only its submatrices
`P`	number of processors P
`S[k][l]`	CS matrix entry s_{kl} – is stored in shared memory
`d[kk]`	constant subvector $\boldsymbol{d}_k$ of the equation system – processor only stores its own subvectors
`iter`	iteration number t
`kk`	loop variable – loops through bodies on the local processor
`l`	loop variable – loops through all bodies
`ll`	loop variable – loops through bodies on the remote processor `p`
`me`	number of the local processor
`y[kk]`	solution subvector $\boldsymbol{y}_k(t+1)$ – processor stores only its own subvectors
`ycom`	buffers solution subvector `yold[l]` in local memory
`yold[k]`	solution subvector $\boldsymbol{y}_k(t)$ – is stored in shared memory
`z[kk][l]`	dot product $\boldsymbol{z}_{kl}$ – processor stores only its dot products

Table 5.2: Objects in the CSM code.

The data distribution in the CSM implementation is significantly different from both other implementations as shown in Table 5.2. The whole solution vector $\boldsymbol{y}(t)$ at time t is stored in the global shared memory, and each processor updates its part of $\boldsymbol{y}(t)$ in every iteration step. In contrast to the other two implementations, $\boldsymbol{y}(t)$ is only stored once, and the processors do not have their own copies, which reduces the amount of memory used. In addition, the CS matrix $\boldsymbol{S}$, the data of the boundary elements, and the data of the bodies are also stored in the global shared memory. The boundary-element data is written in the discretization routine and is accessed read-only during the calculation of the matrices and the iteration. Of course, it has to be communicated to the local caches of the processor whenever it is accessed by this

processor, but since this data is read-only and organized in coherent blocks for each boundary element, it is never invalidated. The only reason it might be flushed from the cache, is when the cache of limited size is full and cache space is needed for other data (*capacity misses*).

However, the matrix $\boldsymbol{C}$, the constant vector $\boldsymbol{d}$, and the solution vector $\boldsymbol{y}(t+1)$ at time $t+1$ are distributed as in the message-passing versions and are stored in the local address spaces of the processors, since they are accessed only locally.

Since we do not overlap computation with communication (it turns out that the communication overhead of the CSM code is negligible), there is no scan of the CS matrix $\boldsymbol{S}$ or any request for remote data at the beginning of the iteration. Instead, the processor loops through the columns and rows of the submatrices, calculates them if they are not stored in memory, accesses the globally stored solution vector using the CICO directives, and updates the dot products as illustrated in Figure 5.11. One check-out copies a coherent block of the size of one cache block from the shared memory to the processor's cache. Assume that we want to get a copy of the subvector $\boldsymbol{y}_k(t)$. If we check out the first element of this vector, this check-out not only copies this first element, but also its succeeding elements. For example, if we have a cache block size of 32 bytes and use double-precision numbers for $\boldsymbol{y}_k(t)$ which have a size of 8 bytes, one cache block contains $32/8 = 4$ elements of $\boldsymbol{y}_k$. Thus, if we check out the first element, the first 4 elements are actually copied to the cache with this one check-out. For this reason, we do not check out every element of $\boldsymbol{y}_k(t)$, but only every fourth element. We use shared check-outs `check_out_S`, since the performance analysis of the program shows that there are fewer cache misses – a cache miss occurs, when data that is assumed to be in the cache is not in the cache

```
INITSOLUTION()  // initialize solution with 0
BARRIER()  // synchronize all processors
/* iteration */
Loop iter = 0..(ITER_MAX-1)
  /* update local solution */
  Loop kk = 1..NP  // loop through local bodies (rows)
    Loop l = 1..N  // loop through all bodies (columns)
      If (iter % S[kk][l] == 0)  // check if to update influence
        /* influence from remote body l onto local body kk must be
           updated */
        If (C[kk][l] is not stored in memory)
          /* sub matrix C_kl is calculated */
          CREATE(C[kk][l])
        End If
        /* check out yold[l] from global shared memory in packages
           according to the cache blocks */
        CHECK_OUT(yold[l])
        ycom = yold[l]  // access global yold and copy it to local ycom
        /* check in yold[l] into global shared memory in packages
           according to the cache blocks */
        CHECK_IN(yold[l])
        /* dot product */
        dot = C[kk][l] . ycom
        y[kk] += dot - z[kk][l]  // update by difference
        z[kk][l] = dot  // store dot product for next update
      End If
    End Loop l
    /* check out yold[(me-1)*NP+kk] from global shared memory in packages
       according to the cache blocks */
    CHECK_OUT(yold[(me-1)*NP+kk])
    yold[(me-1)*NP+kk] = y[kk]  // copy local y[kk] to global yold
    /* check in yold[[(me-1)*NP+kk] into global shared memory in packages
       according to the cache blocks (invalidate) */
    CHECK_IN(yold[[(me-1)*NP+kk])
  End Loop kk
End Loop iter
BARRIER() // synchronize all processors
```

Figure 5.11: Iteration algorithm in CSM.

but has to be fetched from main memory or from shared memory – rather than using exclusive check-outs `check_out_X`. The phase in which more than one processor access the same memory block simultaneously is kept small by fetching the global data at one location, copying it into local memory for further reference, and flushing the associated cache blocks by issuing check-in directives `check_in`. The same idea is used for updating a processor's solution subvector by first updating the solution subvector in local memory, and then copying it to shared memory at the end of the iteration step using `check_out_X` and `check_in`.

In comparison to the other two implementations, mainly the CMMD version, the implementation in CSM is the easiest and the most transparent, and has the fewest lines of code. Keeping read-only data in shared memory reduces the amount of memory and makes the code more transparent and easier to maintain. The CICO model allows us to analyze and optimize the code with respect to shared-memory accesses independent of the program's functionality. Furthermore, the implementation of the CSM code on the Dir_1SW hardware has the most promising performance of all three implementations.

Chapter 6

N Bodies in a Spherical Container

So far we have considered N bodies in an unbounded domain. In this chapter the boundary integral equations (3.57) are modified to describe the double layer densities on N bodies that are placed inside a closed spherical container. The bodies are surrounded by vacuum and are perfect conductors as in the unbounded case. The spherical container is also a perfect conductor, so that its exterior region does not influence the interior system. By using the *method of images* a new Green's function $\bar{G}(\boldsymbol{x}, \boldsymbol{\xi})$ and a new double layer kernel $\bar{K}(\boldsymbol{x}, \boldsymbol{\xi})$ are derived. The domain for the unknown double layer densities on the bodies' surfaces stays the same. Thus, upon descretization the number of unknowns – there is one unknown collocated double layer density per boundary element – stays the same. The fact that there is a spherical container surrounding the bodies is completely accounted for by the modified Green's function and the new double layer kernel. The resulting discretized algebraic equation system is exactly of the same form as the one in Chapter 4, except that the elements in the system matrix $\boldsymbol{A}$ and in the constant

vector $\boldsymbol{b}$ have changed. Thus, the same iteration algorithm and its same implementations can be used. However, the routines for setting up the system matrix $\boldsymbol{A}$ and the vector $\boldsymbol{b}$ have to be modified. The net result of this approach is that the amount of inter-processor communication stays the same, but the computation to communication ratio increases significantly — a good portent for parallelism.

In electrostatics the method of images is a well known trick for matching BCs on boundaries of simple shape by placing virtual point charges in the domain in addition to the physically existing point charges [25]. The boundary conditions on these boundaries determine the magnitudes and the positions of these point charges. To apply the method of images to charged bodies not only virtual point charges but also the image of the double layer is needed. However, the image of the double layer can be derived from the virtual point charges in a straightforward way.

This approach is used to calculate the macroscopic dielectric constants and thermal conductivities of two-phase materials. The N perfectly conducting bodies are inclusions in an isotropic medium which is surrounded by a spherical perfect conductor. For instance, the dielectric constant of this medium is changed by the polarization effect of the inclusions. The final section describes a numerical approach for calculating this dielectric constant by using the dipole moments of the bodies. The same method yields the thermal conductivity of an isolating matrix with N perfectly conducting bodies as inclusions, since the time-independent heat conduction is also described by the Laplace equation.

6.1 Image of a Point Charge

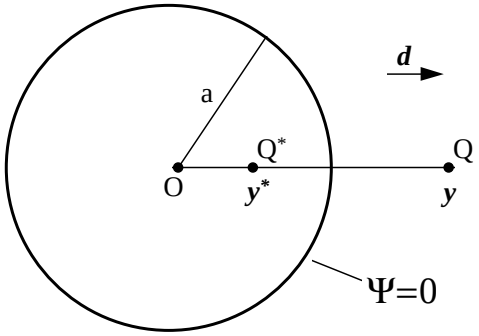

Figure 6.1: Image of a point charge. The point charge is at $\boldsymbol{y}$. Its image is at $\boldsymbol{y}^*$. The spherical container with radius a has zero potential.

First the image of a point charge is derived. Figure 6.1 shows the geometry of a perfect spherical conductor of radius a whose surface S_C is the equipotential $\psi(\boldsymbol{\eta}) = 0$, $\boldsymbol{\eta} \in S_C$, and a point charge outside the conductor. The image of this point charge is sought, so that the potentials of the original and its image charge cancel each other on the boundary S_C. The image of the original point charge could be any distribution of singularities, but it turns out that the image is also a point charge. If $\boldsymbol{y}$ is the position vector of the original point charge and Q is its magnitude, then $\boldsymbol{y}^*$ and Q^* describe the image location and charge. The electric potential created by the original point charge is

$$\psi(\boldsymbol{x}) = \frac{Q}{|\boldsymbol{x} - \boldsymbol{y}|} \tag{6.1}$$

and the unknown image potential is denoted as $\psi^*(\boldsymbol{x})$. The boundary condition on the container surface is: $\psi(\boldsymbol{\eta})+\psi^*(\boldsymbol{\eta}) = 0$. The potential of the original point charge is expanded by Legendre expansion about the center of the spherical container:

$$\psi(\boldsymbol{x}) = \frac{Q}{|\boldsymbol{x}-\boldsymbol{y}|} = Q\sum_{n=0}^{\infty} \frac{|\boldsymbol{x}|^{2n+1}}{R^{n+1}} \frac{(\boldsymbol{d}\cdot\nabla)^n}{n!}\frac{1}{|\boldsymbol{x}|} \ , \tag{6.2}$$

where $R = |\boldsymbol{y}|$ and $\boldsymbol{d} = \frac{\boldsymbol{y}}{R}$. The potential of the image point charge is written as a multipole expansion

$$\psi^*(\boldsymbol{x}) = \frac{Q^*}{|\boldsymbol{x}-\boldsymbol{y}^*|} = \sum_{n=0}^{\infty} A_n \frac{(\boldsymbol{d}\cdot\nabla)^n}{n!}\frac{1}{|\boldsymbol{x}|} \ . \tag{6.3}$$

Considering the boundary condition on the container surface $|\boldsymbol{x}| = a$ a comparison of the coefficients of both sums yields the coefficients A_n:

$$A_n = -Q\frac{a^{2n+1}}{R^{n+1}} \quad ; n = 0, 1, 2, \ldots \ . \tag{6.4}$$

Now we have the solution for the image potential as a multipole expansion, but we are looking for a singularity representation for ψ^*. The symmetry of the problem implies that the image of the point charge has to be a distribution of singularities placed on the straight line connecting $\boldsymbol{y}$ and the sphere center. The location of these singularities is reduced to the line section between the sphere's center and the point $\boldsymbol{y}^*$ in Figure 6.1, which is given by $|\boldsymbol{y}^*| = \frac{a^2}{R}$. This is a guess, but a reasonable one looking at the analogous problem of the image system for the Stokeslet [31]. Let $f(x)$ represent the singularity distribution along this line section, then the image

potential is given by the integral

$$\psi^*(\boldsymbol{x}) = \int_0^{\frac{a^2}{R}} f(\boldsymbol{\xi}) \frac{1}{|\boldsymbol{x} - \xi \boldsymbol{d}|}\, d\xi \ . \tag{6.5}$$

$\frac{1}{|\boldsymbol{x}-\xi\boldsymbol{d}|}$ is expanded as a Taylor series about the sphere's center:

$$\frac{1}{|\boldsymbol{x} - \xi \boldsymbol{d}|} = \sum_{n=0}^{\infty} \xi^n \frac{(\boldsymbol{d} \cdot \nabla)^n}{n!} \frac{1}{|\boldsymbol{x}|} \ . \tag{6.6}$$

Thus,

$$\psi^*(\boldsymbol{x}) = \sum_{n=0}^{\infty} \left(\int_0^{\frac{a^2}{R}} f(\xi) \xi^n \, d\xi \right) \frac{(\boldsymbol{d} \cdot \nabla)^n}{n!} \frac{1}{|\boldsymbol{x}|} \ . \tag{6.7}$$

The comparison of this representation for ψ^* to its multipole expansion, Equation (6.3), yields further constraints for the coefficients A_n:

$$A_n = -Q \frac{a^{2n+1}}{R^{n+1}} = \int_0^{\frac{a^2}{R}} f(\xi) \xi^n \, d\xi \ . \tag{6.8}$$

The integral of Dirac's delta function δ resolves this identity:

$$\int_0^{\frac{a^2}{R}} \delta(\xi - \frac{a^2}{R}) \xi^n \, d\xi = \frac{a^{2n}}{R^n} \ . \tag{6.9}$$

Thus,

$$f(x) = -Q \frac{a}{R} \delta(x - \frac{a^2}{R}) \ . \tag{6.10}$$

This proves, that the distribution of the singularities is exactly one point charge of magnitude $Q^* = -Q\frac{a}{R}$ and at the position $\boldsymbol{y}^* = \boldsymbol{d}\frac{a^2}{R} = \boldsymbol{y}\frac{a^2}{|\boldsymbol{y}|^2}$.

Since the image of a point charge outside the container is also a point charge, we can reverse the problem and look for the image of a point charge inside the container, which is the actual problem we want to solve. Note that this is not a valid approach for the image system of a Stokeslet, since the image of a Stokeslet outside the container is a whole distribution of singularities inside the container [31]. The position of the point charge inside the container is given by $\boldsymbol{y}$ and its magnitude by Q. The equations for its image are the same as in the problem above:

$$\boldsymbol{y}^* = \boldsymbol{d}\frac{a^2}{R} = \boldsymbol{y}\frac{a^2}{|\boldsymbol{y}|^2} \ , \tag{6.11}$$

$$Q^* = -Q\frac{a}{R} = -Q\frac{a}{|\boldsymbol{y}|} \ . \tag{6.12}$$

Since the Green's function for one point charge in an unbounded domain is given by

$$G(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} \ , \tag{6.13}$$

the Green's function for the combination of one point charge and its image is readily given by:

$$\bar{G}(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} - \frac{a}{4\pi|\boldsymbol{\xi}| \left|\boldsymbol{x} - \frac{a^2}{|\boldsymbol{\xi}|^2}\boldsymbol{\xi}\right|} = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} - \frac{a}{4\pi|\boldsymbol{\xi}| \ |\boldsymbol{x} - \boldsymbol{\xi}^*|} \ , \tag{6.14}$$

where $\boldsymbol{\xi}^* = \frac{a^2}{|\boldsymbol{\xi}|^2}\boldsymbol{\xi}$.

6.2 Image of the Double Layer

To apply the method of images to CDLBIEM, the image of the double layer is needed as well. Now a dipole can be obtained by a limiting process of coalescence of two point charges, so the dipole image can be derived in a straightforward way. In mathematical terms, we note that the double layer kernel is obtained as a normal derivative (with respect to $\boldsymbol{\xi}$) of the Green's function:

$$K(\boldsymbol{x}, \boldsymbol{\xi}) = 2\nabla_{\boldsymbol{\xi}} G(\boldsymbol{x}, \boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi}) \ . \tag{6.15}$$

The same operations are now applied to the image terms. We start with

$$\bar{G}(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi} \left[\frac{1}{r} - \frac{\alpha}{r^*} \right] \ , \tag{6.16}$$

where

$$\alpha = \frac{a}{|\boldsymbol{\xi}|} = \frac{|\boldsymbol{\xi}^*|}{a} \ ,$$

$$r = |\boldsymbol{x} - \boldsymbol{\xi}| = \sqrt{(x_i - \xi_i)(x_i - \xi_i)} \ , \quad r^* = |\boldsymbol{x} - \boldsymbol{\xi}^*| = \sqrt{(x_i - \xi_i^*)(x_i - \xi_i^*)} \ .$$

Now use the following expressions for differentiation with respect to $\boldsymbol{\xi}$,

$$\frac{\partial}{\partial \xi_i} \frac{1}{r} = \frac{x_i - \xi_i}{r^3} \ , \tag{6.17}$$

$$\frac{\partial \alpha}{\partial \xi_i} = -\frac{a}{|\boldsymbol{\xi}|^3} \xi_i \ , \tag{6.18}$$

$$\frac{\partial \xi_j^*}{\partial \xi_i} = \frac{a^2}{|\boldsymbol{\xi}|^2}\left(\delta_{ij} - 2e_i e_j\right) \quad . \tag{6.19}$$

Here the unit vector $\boldsymbol{e}$ is defined as $e_i = \frac{\xi_i}{|\boldsymbol{\xi}|}$. Using these derivatives the expression for the new double layer kernel is obtained:

$$\bar{K}(\boldsymbol{x},\boldsymbol{\xi}) = \frac{1}{2\pi}\hat{n}_i \left\{ \frac{x_i - \xi_i}{|\boldsymbol{x}-\boldsymbol{\xi}|^3} + \frac{|\boldsymbol{\xi}^*|^2}{|\boldsymbol{x}-\boldsymbol{\xi}^*|a^3}\left[-|\boldsymbol{\xi}^*|(\delta_{ij} - 2e_i e_j)\frac{\boldsymbol{x}_j - \boldsymbol{\xi}_j^*}{|\boldsymbol{x}-\boldsymbol{\xi}^*|^2} + e_i \right] \right\} \quad . \tag{6.20}$$

Note, that the first term of $\bar{K}(\boldsymbol{x},\boldsymbol{\xi})$ is the contribution of the original double layer and its second term is all contributed from its image. Note also, that the boundary integral equations, such as (3.57), stay the same, except that the contribution of the image point charge has to be added in the kernel of the double layer integral. Thus, the boundary integral equation (3.57) turns into:

$$\begin{aligned} \varphi(\boldsymbol{\eta}) &= -\psi^{\infty}(\boldsymbol{\eta}) - \frac{1}{S_k}\oint_{S_k} \varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \\ &- \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{\eta}-\boldsymbol{x}_l|} - \frac{aQ_l}{|\boldsymbol{x}_l|\left|\boldsymbol{\eta} - \frac{a^2}{|\boldsymbol{x}_l|^2}\boldsymbol{x}_l\right|} + \oint_{S_l} \bar{K}(\boldsymbol{\eta},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\boldsymbol{\xi}} \right\} \end{aligned} \tag{6.21}$$

$$\boldsymbol{\eta} \in S_k \qquad ; k = 1...N \quad .$$

The integration domain of the double layer integral is unchanged. The discretization and numerical treatment of this modified boundary integral equation is the same as for the original boundary integral equation. This ensures that only minor changes have to be made in the program codes, namely in the routines that calculate the system matrix $\boldsymbol{A}$ and the constant vector $\boldsymbol{b}$.

6.3 Dielectric Constant

The method described in the previous sections provides an efficient numerical approach for solving the electric potential of N bodies as inclusions in a container. This method may be extended to calculate the effective dielectric constant of these perfectly conducting inclusions in a void[1] matrix according to the lines in [4].

The *dielectric constant* is a 2nd order tensor $\boldsymbol{\epsilon}$ in the general case of anisotropic inclusions. It describes the relation between the macroscopic *electric displacement* $\langle \boldsymbol{D} \rangle$ and the superimposed *electric field* $\langle \boldsymbol{E} \rangle$:

$$\langle \boldsymbol{D} \rangle = \boldsymbol{\epsilon} \cdot \langle \boldsymbol{E} \rangle \ . \tag{6.22}$$

The brackets $<$ and $>$ indicate that the quantities are volume averages over the whole domain of the container.

In order to derive an expression for $\boldsymbol{\epsilon}$, we start with the multipole expansion of the double layer potential for one body k around the point $\mathbf{0}$

$$\begin{aligned}
\psi(\boldsymbol{x}) &= \oint_{S_k} K(\boldsymbol{x},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_\xi \\
&= \oint_{S_k} 2\nabla_\xi G(\boldsymbol{x},\boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_\xi \\
&= \oint_{S_k} 2\hat{\boldsymbol{n}}(\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_\xi \cdot \nabla_\xi G(\boldsymbol{x},\boldsymbol{\xi})|_{\xi=\mathbf{0}} + \ldots \\
&= \underbrace{\oint_{S_k} \frac{1}{2\pi}\hat{\boldsymbol{n}}(\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_\xi}_{=\boldsymbol{p}_k} \cdot \nabla_\xi \frac{1}{|\boldsymbol{x}-\boldsymbol{\xi}|}\bigg|_{\xi=\mathbf{0}} + \ldots
\end{aligned} \tag{6.23}$$

[1] The analysis is readily extended to a dielectric matrix by introducing the relative electric permittivity.

where $\boldsymbol{p}_k$ is the dipole moment of body k and is evaluated by using the double layer density, which is returned by CDLBIEM as described in Chapter 5.

The electric polarization $\langle \boldsymbol{P} \rangle$ of N bodies is the volume average of the dipole moments over all bodies

$$\langle \boldsymbol{P} \rangle = \frac{1}{V_R} \sum_{k=1}^{N} \boldsymbol{p}_k \ . \tag{6.24}$$

V_R is the volume of the container including the bodies. The definition of the electric displacement is

$$\langle \boldsymbol{D} \rangle = \langle \boldsymbol{E} \rangle + 4\pi \langle \boldsymbol{P} \rangle \ . \tag{6.25}$$

By substituting $\boldsymbol{\epsilon}$ with $\boldsymbol{\delta} + c\boldsymbol{\beta}$, we rewrite Equation (6.22) as

$$\langle \boldsymbol{D} \rangle = (\boldsymbol{\delta} + c\boldsymbol{\beta}) \cdot \langle \boldsymbol{E} \rangle \ . \tag{6.26}$$

$\boldsymbol{\delta}$ is the unit matrix, $\boldsymbol{\beta}$ the *electric susceptibility*, and c the volume fraction of the bodies:

$$c = \frac{1}{V_R} \sum_{k=1}^{N} V_k \ . \tag{6.27}$$

The comparison of Equation (6.26) to Equation (6.25) yields an expression for the electric susceptibility:

$$\boldsymbol{\beta} \cdot \langle \boldsymbol{E} \rangle = \frac{4\pi}{V_R c} \sum_{k=1}^{N} \boldsymbol{p}_k \ . \tag{6.28}$$

The tensor $\boldsymbol{\beta}$ has 9 unknowns. By running the algorithm three times, each time setting the

electric field $\langle \boldsymbol{E} \rangle$ to a different unit vector $\boldsymbol{e}_j$, the 9 unknowns are obtained by

$$\beta_{ij} = \frac{4\pi \left(\sum_{k=1}^{N} \boldsymbol{p}_k \right)_i}{V_R c e_j} \ . \tag{6.29}$$

The dielectric constant is then given by

$$\boldsymbol{\epsilon} = \boldsymbol{\delta} + c\boldsymbol{\beta} \ . \tag{6.30}$$

This analysis may also be used to obtain the thermal conductivity of an inclusion of N perfect thermal conductors in a conducting matrix, which is surrounded by a perfectly conducting container. Assuming that the heat conduction in the matrix obeys Fourier's law and the isotropic matrix has the constant thermal conductivity k_1, the heat flux in the matrix is

$$\boldsymbol{J}(\boldsymbol{x}) = -k_1 \nabla T(\boldsymbol{x}) \tag{6.31}$$

where T is the temperature in the matrix. From the energy balance we obtain the Laplace equation for the temperature, the same equation which describes the electric potential:

$$\nabla^2 T(\boldsymbol{x}) = 0 \ . \tag{6.32}$$

We look for the thermal conductivity $\boldsymbol{k}$, which is (analogous to the dielectric constant) also a 2nd order tensor in the anisotropic case, and which describes the relation between the heat flux

dielectric constant	thermal conductivity
$\boldsymbol{\epsilon}$	$\boldsymbol{k}$
$\boldsymbol{\beta}$	$\boldsymbol{\beta}$
$\boldsymbol{E}$	$-\nabla T$
$\boldsymbol{D}$	$\boldsymbol{J}$
ψ	T
$\boldsymbol{P}$	$\boldsymbol{P}$
$\boldsymbol{p}_k$	$\boldsymbol{p}_k$

Table 6.1: Correspondence between the dielectric constant and the thermal conductivity problems.

of the whole system and a superimposed temperature gradient:

$$\langle \boldsymbol{J} \rangle = -\boldsymbol{k} \cdot \langle \nabla T \rangle \ . \tag{6.33}$$

The problem for the thermal conductivity and the problem for the dielectric constant are identical. The quantities for both problems correspond as listed in Table 6.1.

The parameter $\boldsymbol{\beta}$ for the thermal conductivity problem is identical to $\boldsymbol{\beta}$ for the dielectric constant problem:

$$\boldsymbol{\beta} \cdot \langle -\nabla T \rangle = \frac{4\pi}{V_R c} \sum_{k=1}^{N} \boldsymbol{p}_k \tag{6.34}$$

and the thermal conductivity is given as

$$\boldsymbol{k} = k_1 \boldsymbol{\delta} + c\boldsymbol{\beta} \ . \tag{6.35}$$

Chapter 7

Performance Analysis and Results

In this section the performance of the completed double layer boundary integral method CDLBIEM, the parallel implementation of the iteration algorithm, and the performance of the different programming models are discussed. Furthermore, results for N bodies as inclusions in a spherical container are presented. In Section 7.1 the result obtained by CDLBIEM is verified to be numerically accurate. In the subsequent section the performance (execution time) of the parallel iteration algorithm as a function of the CS matrix is investigated for the message-passing program. The communication in the Split-C program is shown to be faster than in the message-passing program. Then, the scaleup of the algorithm is investigated using the CSM implementation. Finally, results for N bodies as inclusions in a spherical perfect conductor and the dielectric constants and the thermal conductivities of those systems are presented.

7.1 Accuracy of CDLBIEM

CDLBIEM is a numerical method that includes a numerical error. This section investigates the reliability of the results obtained by CDLBIEM. A direct comparison with experimental data is not appropriate for these model calculations. However, a detailed comparison with established analytic or numerical results is possible for special geometries. First, the CDLBIEM solution is compared to the analytic solution in the case of two equally sized and equally charged spheres. Second, the CDLBIEM solution is compared to the boundary collocation solution for 32 equally sized and equally charged spheres placed on a straight line.

The first comparison uses the analytic solution for two spheres, which is readily available in bispherical coordinates. The data of the spheres for this comparison is listed in Table 7.1. The electric potential on both spheres' surfaces calculated by CDLBIEM is compared to the analytic solution in Figure 7.1. The key error of CDLBIEM is caused by the discretization of the surfaces by boundary elements. Since the interaction between the almost touching spheres is very strong, CDLBIEM with four boundary elements has a large relative error. However, this error is decreased by three orders of magnitude by using 320 boundary elements and CDLBIEM gives a good approximation for the solution.

The second test compares CDLBIEM's result to the boundary collocation solution. In order to use the boundary collocation method, all 32 spheres are placed on a straight line with equal distances as described in Table 7.2. The agreement between both methods is good and improves with boundary element mesh refinement as shown in Figure 7.2.

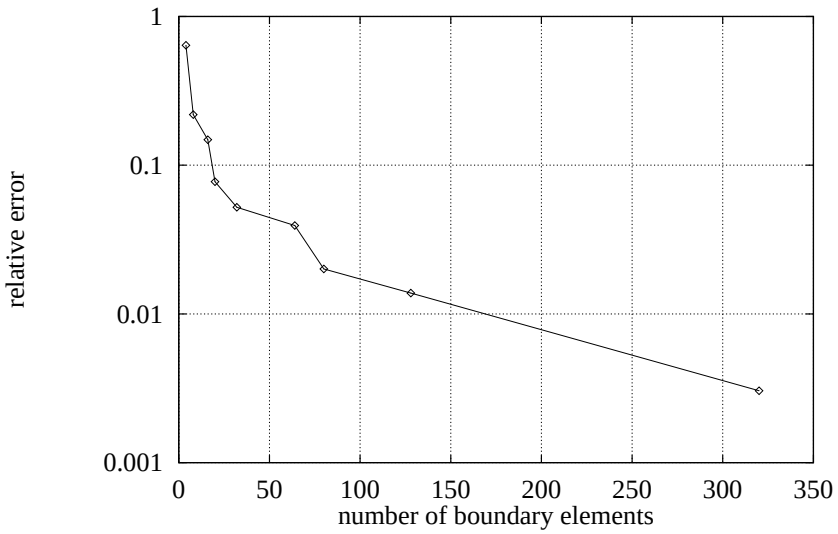

Figure 7.1: Numerical error of CDLBIEM for two spheres. The relative error of the electric potential on two equally sized and equally charged spheres versus different numbers of boundary elements.

number of spheres	2
radii of the spheres	1.0
distance between the spheres' centers	2.2
distance between the spheres' surfaces	0.2
net charge of one sphere	1.0
no superimposed electric field	

Table 7.1: Input data for numerical error of CDLBIEM for two spheres. Data of the two spheres used to compare CDLBIEM to the analytic solution.

number of spheres	32
radii of the spheres	1.0
distance between the spheres' centers	2.5
distance between the spheres' surfaces	0.5
net charge of one sphere	1.0
no superimposed electric field	

Table 7.2: Input data for numerical error of CDLBIEM for 32 spheres. Data of the 32 spheres used to compare CDLBIEM to the boundary collocation method.

7.2 Performance of the CMMD Implementation

This first implementation is mainly used to analyze the asynchronous iteration algorithm and its speedup as a function of the CS matrix. A detailed scaleup analysis has not been feasible, yet, since the CM-5 partition we used has only 32 nodes and is too small for an exhaustive scaleup analysis. Thus, detailed scaleup analyses are limited to the CSM implementation.

To illustrate the problems which have been solved with the CMMD implementation an array of 256 spheres is illustrated in Figure 7.3 with the input data as in Table 7.3. The spheres are randomly distributed in a 3-dimensional cube; their graylevel represents their surface potential ranging from light-gray (low potential) to dark-gray (high potential). In this example, there is

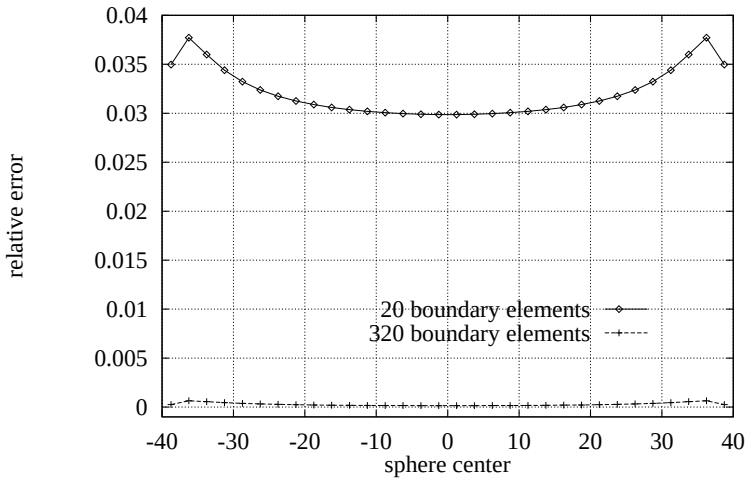

Figure 7.2: Numerical error of CDLBIEM for 32 spheres. The relative error of the electric potential on 32 equally sized and equally charged spheres versus the position of the spheres.

no superimposed ambient electric field and the electric potential is generated only by the net charges on the spheres, and therefore, the highest potential is near the center of the cube.

As anticipated in Section 5.4.3, we expect a tremendous speedup of the execution by reducing the frequency of interaction updates. The problem, which leads to the result in Figure 7.3, has been run with different CS matrices. Reducing the frequency of updates as given *a priori* in the CS matrix, is expected to reduce the amount of computation and communication and still lead to the correct solution. Although the data for 256 spheres fits into the memory of the CM-5 (see Table 7.4) even when all submatrices are permanently stored in memory, in order to test the performance of the program and to extrapolate its behavior to larger problem sizes, only a part of the submatrices is stored, while the rest is calculated during the execution of

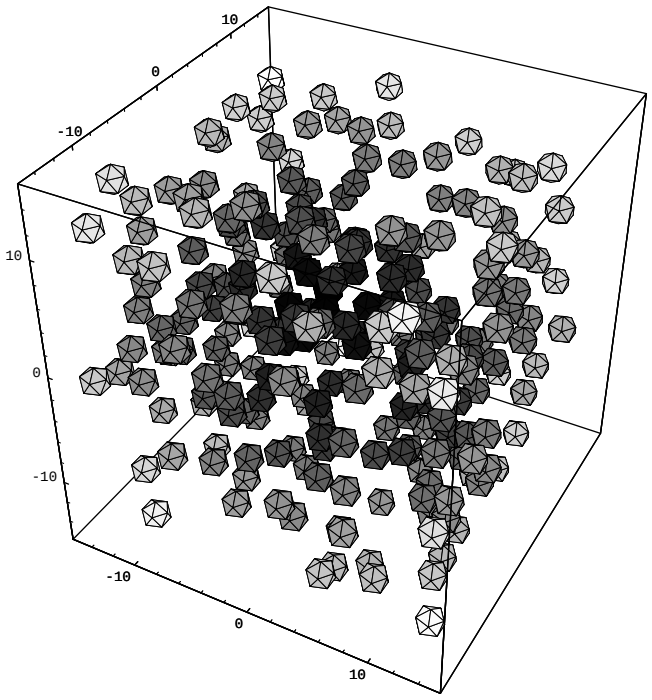

Figure 7.3: Electric potential of 256 spheres, which are randomly distributed in a cube. Light-gray corresponds to low potential and dark-gray corresponds to high potential.

number of spheres	256
radii of the spheres	1.0
minimal distance between the spheres' surfaces	0.1
net charge of one sphere	1.0
no superimposed electric field	
domain: cube with sidelength	30
number of boundary elements per sphere	20
number of processors	32

Table 7.3: Input data for 256 spheres.

CM-5 partition size	32 processors
DRAM on one node	32 MByte
vector units	not used
message-passing library	CMMD 3.0
C compiler	gcc

Table 7.4: Machine parameters for the CMMD runs.

the iteration, as discussed in Section 5.4.4. There are two sets of runs: The first set, where half of the matrix is stored in memory and the other half is calculated. The second set has one quarter of the matrix stored and the other three quarters are calculated. The first set includes 5 runs with different CS matrices, where the maximal entry in the CS matrix is either set to 1, 10, 20, 40, or 80. This maximal entry corresponds to the interaction between the two most distant bodies. The other entries are fractions of this maximal entry, where one fraction is the ratio of the distance between the corresponding bodies and the maximal distance occuring in the system. The second set, where one quarter of the matrix is stored, includes runs with 1, 40, and 80 for the maximal entry.

The less frequent updates of the solution subvectors and the partial dot products lead to

more iteration steps needed for convergence. Suppressing updates basically sets submatrices in the system matrix $\boldsymbol{C}$ to zero, which might increase the spectral radius of $\boldsymbol{C}$. Since the eigenvalues and the spectral radius of $\boldsymbol{C}$ are dominated by its diagonal submatrices, which are never set to zero, the matrix $\boldsymbol{C}$ is still stable (its spectral radius is less than one), and the iteration still converges.

For instance, $\boldsymbol{C}$ for the problem of two spheres in an unbounded domain (see Figure 7.11) has a spectral radius of 0.296. By setting any of the two off-diagonal submatrices, which describe the interaction between the two bodies, to zero, the spectral radius decreases to 0.178.

Figure 7.4 shows the relative error of the double layer density versus the iteration number for the 5 different runs, when half of the matrix is permanently stored in memory. This relative error is not based on the accurate solution, but on the converged solution of a prior run with a high number of iterations. Thus, the relative error does not include the systematic error of CDLBIEM that is discussed in Section 7.1. For this reason, the asymptotic relative error is of the order $O(10^{-16})$, which corresponds to the numerical accuracy of double-precision floating-point numbers (8 Bytes). As anticipated, the number of iteration steps needed for convergence increases, when the frequency of interaction updates decreases.

However, the execution time of the iteration is reduced. The interactions between far apart bodies do not contribute as much to the convergence of the solution as the interactions between near bodies do. Thus, updating the communication and the computation for these far-body interactions is more expensive than suppressing them. While these far-body interactions are being suppressed, the iterative solution converges to a "local" fixed point and waits until "global"

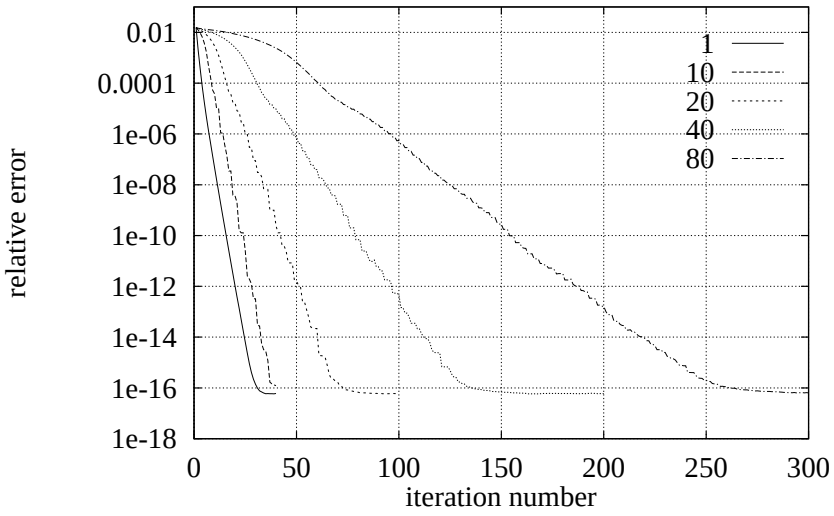

Figure 7.4: Relative error versus iteration number. Half of the system matrix $\boldsymbol{C}$ is stored. One error curve represents one run for one CS matrix with its maximal entry $\in \{1, 10, 20, 40, 80\}$.

information arrives that nudges it towards the "global" fixed point, the final solution of the iteration. This behavior is observed in the step-like shape of the error curves in Figure 7.4, which shows the relative error versus the execution time of the iteration, when half of the matrix is permanently stored in memory. There is a trade-off between the reduced communication and computation time and the increased number of iteration steps: The total execution time is the product of the decreased execution time spent in one iteration step and the increased numbers of iteration steps. By over-suppressing the updates, the total execution time increases. For this reason there is an optimal CS matrix. As can be seen in Figure 7.5, the optimal execution time is obtained by setting the maximal entry of the CS matrix to a value around 40.

In the case, where only a quarter of the matrix is stored, even a better improvement in the

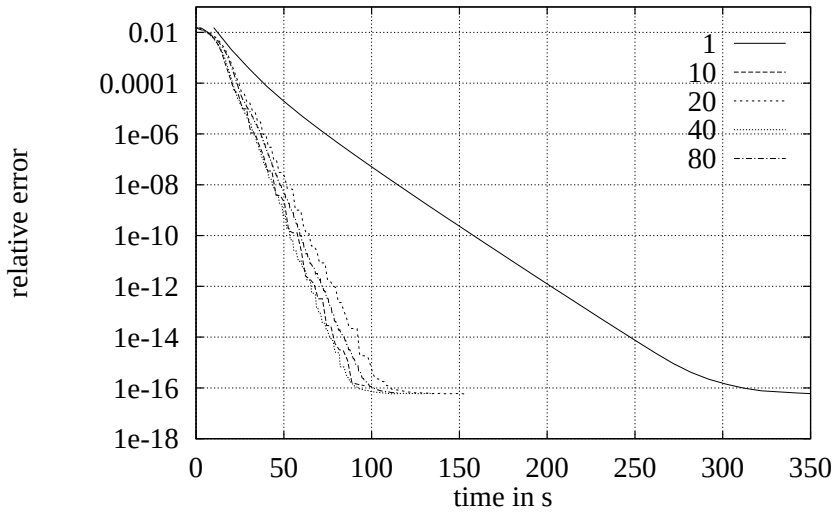

Figure 7.5: Relative error versus execution time in sec. Half of the system matrix $\boldsymbol{C}$ is stored. One error curve represents one run for one CS matrix with its maximal entry $\in \{1, 10, 20, 40, 80\}$.

execution time is observed. Figure 7.6 compares the number of iterations needed for convergence to the number of iterations in the case, where half of the matrix is stored. The relative error versus the iteration number is the same in both cases, since the frequency of the updates is the same. However, since there is twice as much time spent in setting-up the submatrices, one of these updates also takes longer and leads to the comparison in Figure 7.7. The results in this figure are very promising, since the execution times in the case, where a quarter of the matrix is stored, get close to the execution times in the case, where half of the matrix is stored. This is of great advantage for huge problem sizes, where only a small part of the matrix can be stored in memory.

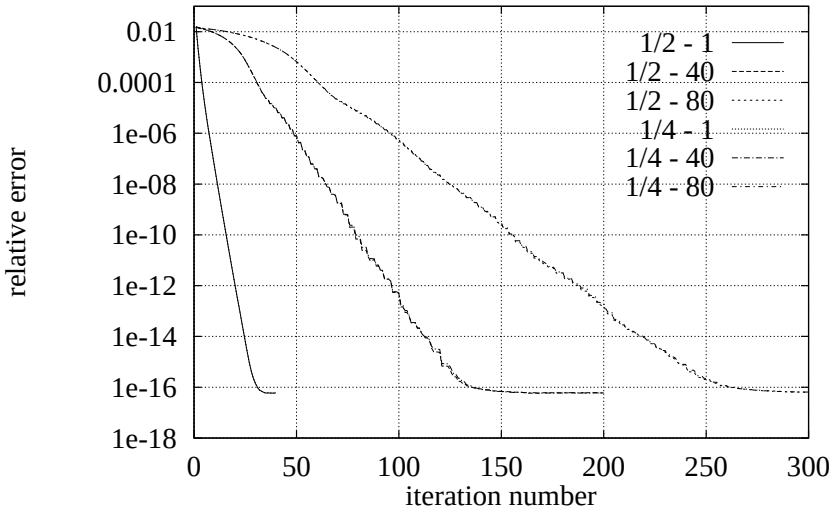

Figure 7.6: Relative error versus iteration number. Half (1/2) or a quarter (1/4) of the system matrix $\boldsymbol{C}$ is stored. One error curve represents one run for one CS matrix with its maximal entry $\in \{1, 40, 80\}$.

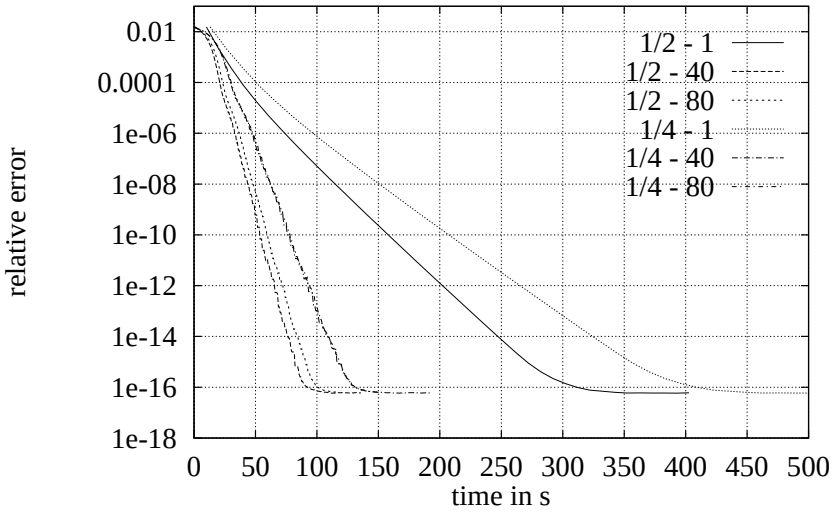

Figure 7.7: Relative error versus execution time in sec. Half (1/2) or a quarter (1/4) of the system matrix $\boldsymbol{C}$ is stored. One error curve represents one run for one CS matrix with its maximal entry $\in \{1, 40, 80\}$.

7.3 Performance of the Split-C Implementation

The performance of the Split-C implementation is very similar to the performance of the CMMD implementation. Split-C provides a static shared-memory programming model and its underlying communication is based on active messages. Its performance is directly related to this message-passing and the results presented in Section 7.2 are not repeated. Instead, the iteration is broken down into four phases: the communication, the matrix calculation, the dot product, and the arranging of output data. The three times of main interest are the overall time spent in the iteration, the time spent in the communication, and the time spent in the matrix calculation. The other two times are small compared to these times.

Table 7.6 compares these times between the CMMD and the Split-C implementation using the parameter set listed in Table 7.5. Note the significant decrease in communication time in the Split-C implementation as compared to the CMMD implementation. This is due to the efficient implementation of the message-passing layer using active messages as discussed in Section 5.6.2. However, the decrease in the overall iteration time is not as big. We believe this is due to the optimizer in the Split-C compiler not being as good as that in the "gcc" compiler on the CM-5 (a problem that can be corrected) [50].

7.4 Performance of the CSM Implementation

This section presents scaling results of the CSM implementation. First of all, the underlying hardware is discussed. The CSM code is run on a cache-coherent shared-memory parallel

number of spheres	256
radii of the spheres	1.0
minimal distance between the spheres' surfaces	0.1
net charge of one sphere	1.0
no superimposed electric field	
domain: cube with sidelength	30
number of boundary elements per sphere	20
number of iterations	40
maximal entry in CS matrix	5
number of processors	32

Table 7.5: Input data for comparison of Split-C code to CMMD code.

Phase	**Split-C**	**CMMD**
40 iterations	108	149
communication	3.5	75
matrix calculation	68	68

Table 7.6: Comparison of Split-C code to CMMD code. Execution time in sec for the whole iteration, the communication phase, and the matrix-calculation phase. The shown execution times are the maxima over all processors.

machine. An architecture which closely fits the CSM programming model is a distributed-memory machine with the cache-coherence protocol Dir_1SW. These results are also presented in our paper [51].

7.4.1 Dir_1SW: An Implementation of CSM

Although the underlying hardware has distributed memory similar to the architecture in Figure 5.1, the Dir_1SW protocol provides a shared-memory model on the programming level. Dir_1SW maintains cache-coherence, which prevents a processor from reading a cached copy of data, when another processor has updated the original memory location. Dir_1SW logically

associates a small amount of memory – called a *directory entry* – with each cache-block-sized piece of the memory. Dir_1SW uses this directory entry to record that the block is either idle (no cached copies), one writable copy exists, or to store the number of read-only copies. The cache coherence protocol uses a combination of hardware and low-level runtime software (trap handlers) to insure that a processor can obtain a copy of a block regardless of the block's prior state. Dir_1SW is designed so that the transitions favored by CICO have the highest performance.

All the results presented in this section use the specific implementation assumptions for Dir_1SW listed in Table 7.7.

7.4.2 The Wisconsin Wind Tunnel

One challenge for performing experiments on the Dir_1SW computer is that it does not exist, yet. For this reason, the measurements in this paper are performed on the *Wisconsin Wind Tunnel (WWT),* a virtual prototype for cache-coherent, shared-memory computers [46]. WWT runs parallel shared-memory programs on a parallel message-passing computer (a Thinking Machines CM-5 [48]) and uses a distributed, discrete-event simulation to concurrently calculate the programs' execution times on a proposed target computer. Wherever possible, WWT exploits the similarities between Dir_1SW and the CM-5 to run faster.

Each processor in the simulated Dir_1SW computer (the target system) executes SPARC binaries. The execution time for each instruction is fixed. Instruction fetches and stack accesses require no additional cycles beyond the basic instruction time. Other memory locations are cached in a node's cache. A cache hit takes no additional cycles, while a cache miss invokes a

processors	8-256 SPARCs
cache	256 KB, 4-way set-associative
block size	32 bytes
TLB	64 entries, fully associative, FIFO replacement
page size	4 KB
message latency	100 cycles remote, 10 cycles to self
barrier latency	100 cycles from last arrival
cache miss	19 cycles + 5 if block is replaced + 8 if replaced block was exclusive copy
cache invalidate	3 cycles + 5 if block is invalidated + 8 if invalidated block was exclusive copy
`check_out`	Same as cache miss, plus 1 cycle for `check_out` issue
`check_in`	Same as cache invalidate, plus 1 cycle for `check_in` issue
directory	10 cycles + 8 if cache block is received + 5 if message is sent + 8 if cache block is sent
trap	255 cycles + 5 for each message sent + 8 for each block sent (directory hardware locked out for first 55 cycles)

Table 7.7: Dir_1SW assumptions.

coherence protocol that sends messages, accesses a directory entry, etc. Each message, cache or directory transition has a cost. Caches and directories process messages in first-come-first-serve order. Queuing delay is included in the cost of a cache miss. Network topology and contention are ignored, and all messages are assumed a fixed latency.

There are two important drawbacks to using WWT. First, when WWT predicts the performance of the proposed Dir_1SW computer, it actually runs programs about 100 times slower [46]. This means that full-scale data sets cannot be used in practice. Second, any model of a real physical system may fail to include key aspects of that system.

WWT, however, provides several advantages for this research. First, we can perform the research prior to the existence of a Dir_1SW computer. Second, we can easily vary system parameters, such as the number of processors and the cache size. The number of processors can even exceed the number of physical processors on the CM-5, so that several virtual processors are simulated on one physical processor. Third, the performance predictions are repeatable and indicate where the execution time is spent. Fourth, virtual prototypes for other computers can be built and the results can be compared in controlled experiments.

7.4.3 Speedup

In order to analyze the scaling of the parallel iteration algorithm, we evaluate the speedup of its CSM implementation according to the lines in Section 5.2.1. We solve a problem with 256 bodies, which is too large for the memory and speed of one processor. However, it can be solved on 16 processors and we use a modified speedup based on the execution time $T_{16}(N)$ on these 16 processors instead of the sequential execution time $T^*(N)$:

$$\bar{S}_P(N) = \frac{T_{16}(N)}{T_P(N)} \ .$$

For an ideal parallel implementation of an algorithm the speedup is equal to the number of processors: $S_P(N) = P$ or $\bar{S}_P(N) = P/16$. Using the constant input set listed in Table 7.8 the iteration algorithm has been executed on 16, 32, 64, 128 and 256 processors. The resulting speedup is shown in Figure 7.8 and is quite close to the ideal straight line. The speedup of the run on 128 processors compared to the run on 16 processors is 7.74. However, the run on

256 processors has a speedup of 12.37 (the ideal speedup is 16). The scaling of the program breaks down on 256 processors since the load balancing routine has no effect for this run. Each processor computes only one body so that there is no degree of freedom in distributing the bodies. With decreasing numbers of bodies per processor, the degrees of freedom decrease, the load balance deteriorates (see Table 7.9), and the scaling of the program gets worse. In production runs we will have a large number of bodies per processor, so that this effect will become less important.

number of processors	16, 32, 64, 128, 256
no container	
number of bodies	256
number of boundary elements per body	20
stored part of system matrix	240/256
maximal number of suppressed updates $\max_{k,l} s_{kl}$	5

Table 7.8: Input data for speedup. 240/256 of the system matrix is stored.

P	**fastest processor**	**slowest processor**	**load balance**	**speedup**
16	413487228	420940543	1.0180	1.0
32	206406117	213151123	1.0327	1.9748
64	102472027	108515008	1.0590	3.8791
128	51451454	54358375	1.0565	7.7438
256	21075391	34020965	1.6143	12.373

Table 7.9: Load balance and speedup for the runs on 16, 32, 64, 128, and 256 processors. The execution time of the iteration is given in clock cycles for the "slowest" and the "fastest" processor. *Load balance* is the ratio of *slowest processor* to *fastest processor*. *Speedup* is based on *slowest processor*.

7.4.4 Time-Constraint Scaleup

As discussed in Section 5.2.1, the time-constraint scaleup of the CSM code is evaluated. The calculation of the system matrix, which has $(NM)^2$ elements, is of the order $O((NM)^2)$. Since the time spent in setting-up the system matrix dominates the execution time of the iteration, this execution time is estimated *a priori* to behave as $O((NM)^2)$.

According to this estimation the input sets for the runs on the different partition sizes are chosen as listed in Table 7.10. The rest of the parameters, for instance the number of boundary elements M, are chosen as listed in Table 7.8. Because of the limitations for the partition sizes there are two different sets of runs, both as presented in Figure 7.9. Both sets show the

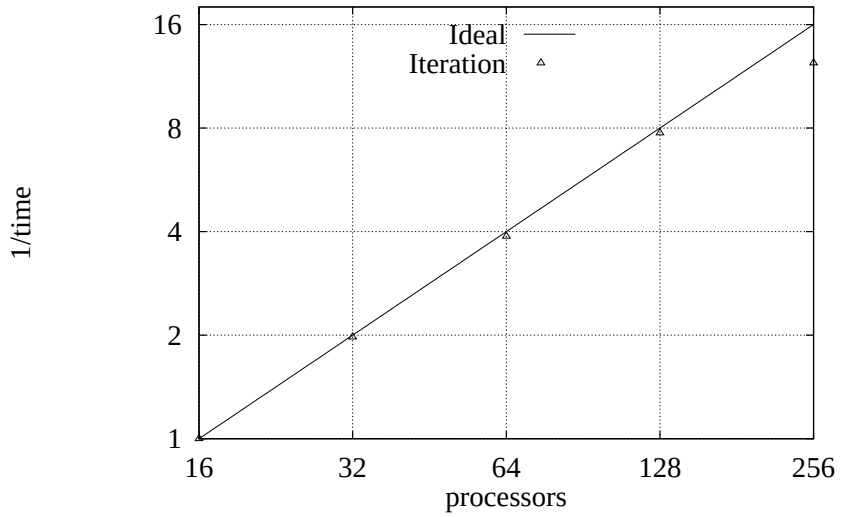

Figure 7.8: Speedup w.r.t. 16 processors. 240/256 of the system matrix is stored.

almost perfect scaleup of the CSM code. However, the scaleup is much worse for the run on 256 processors. This is due to the effect already discussed in Section 7.4.3, where the degree of freedom in distributing the bodies to get a good load balance is zero.

number of processors	number of bodies	stored part
8	64	60/64
32	128	120/128
128	256	240/256
16	64	60/64
64	128	120/128
256	256	240/256

Table 7.10: Input data for time-constraint scaleup.

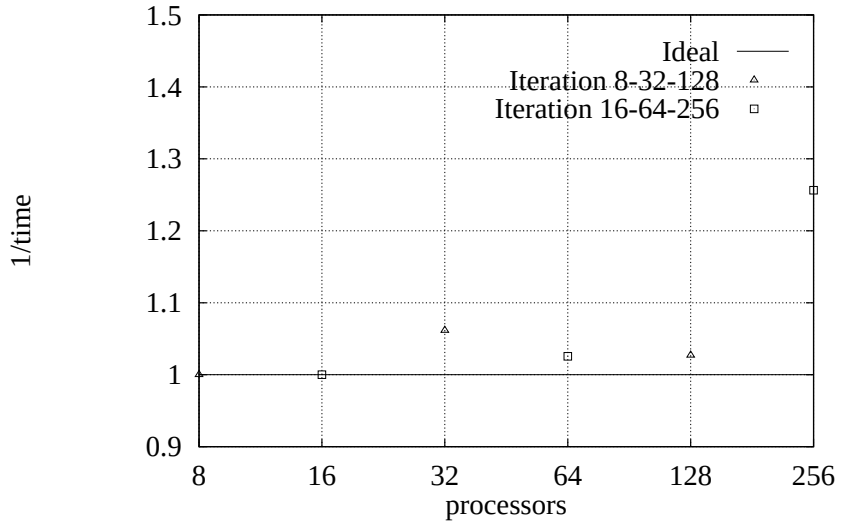

Figure 7.9: Time-constraint scaleup. The curve "8-32-128" is the scaleup of the runs on 8, 32, and 128 processors. The curve "16-64-256" is the scaleup of the runs on 16, 64, and 256 processors.

7.5 N Bodies as Inclusions in a Container

N perfectly conducting spheres are embedded inside a spherical perfectly conducting shell. This section shows results for the electric potential in the vacuum surrounding the inclusions and the corresponding dielectric constants. As derived in Section 6, the computation also yields the thermal conductivity.

7.5.1 Electric Potential

After the double layer density on all body surfaces has been calculated by the parallel iteration algorithm, the electric potential on the bodies and in the vacuum surrounding the bodies is readily available by evaluating the integrals in Equation (3.50) and Equation (3.54). In order to illustrate the equipotential lines, the potential is calculated on planar cross section through the 3-dimensional space.

Since the potential created by one charged sphere in an unbounded infinite domain is given by

$$\psi(\boldsymbol{x}) = \frac{Q}{|\boldsymbol{x}|} \ , \tag{7.1}$$

the equipotential surfaces are spheres and in the cross sections circles. Now place this sphere as an eccentric inclusion within a perfectly conducting spherical container. The potential lines are the thin lines as depicted in Figure 7.10. The thick lines represent boundaries of the container and the included sphere. The data of the spheres for this result is listed in Table 7.11. Note the high density of equipotentials in the upper right and the upper left corners of Figure 7.10,

radius of container	2.0
potential on container's surface	0.0
center of container	$(0.0, 0.0, 0.0)$
radius of included sphere	1.0
net charge of included sphere	1.0
center of included sphere	$(0.0, 0.0, 0.1)$
number of BEs per included sphere	320
no ambient electric field	
normal vector of planar crossection	$(0.0, 1.0, 0.0)$

Table 7.11: Input data for "An eccentric sphere inside a spherical conducting shell".

no container	
radii of spheres	1.0
net charge of one sphere	1.0
center of 1. sphere	$(0.0, 0.0, -1.1)$
center of 2. sphere	$(0.0, 0.0, 1.1)$
number of BEs per sphere	320
no ambient electric field	
normal vector of planar crossection	$(0.0, 1.0, 0.0)$

Table 7.12: Input data for "Two spheres in an unbounded domain".

which is due to the image of the included sphere. This potential field outside the spherical container is not of physical relevance. The real exterior equipotential surfaces are spheres, since the spherical container is a perfect conductor and its interior does not influence its exterior.

Figures 7.11 and 7.12 compare the equipotential lines of two spheres in an unbounded domain to the equipotential lines of two spheres included in a spherical container. The data for these problems is listed in Table 7.12 and Table 7.13.

Figure 7.13 shows the potential field generated by 64 charged spheres as inclusions in a spherical perfect conductor. The spheres are randomly distributed inside the conductor. Note

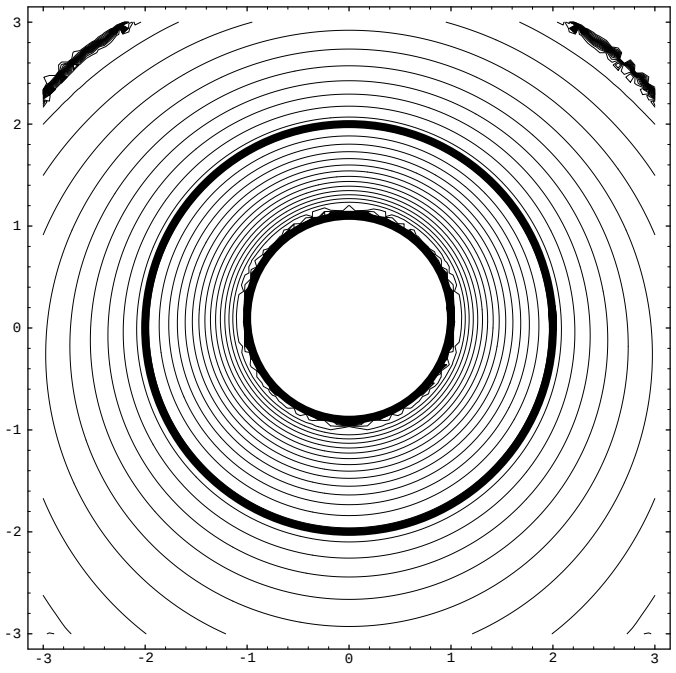

Figure 7.10: An eccentric sphere inside a spherical conducting shell.

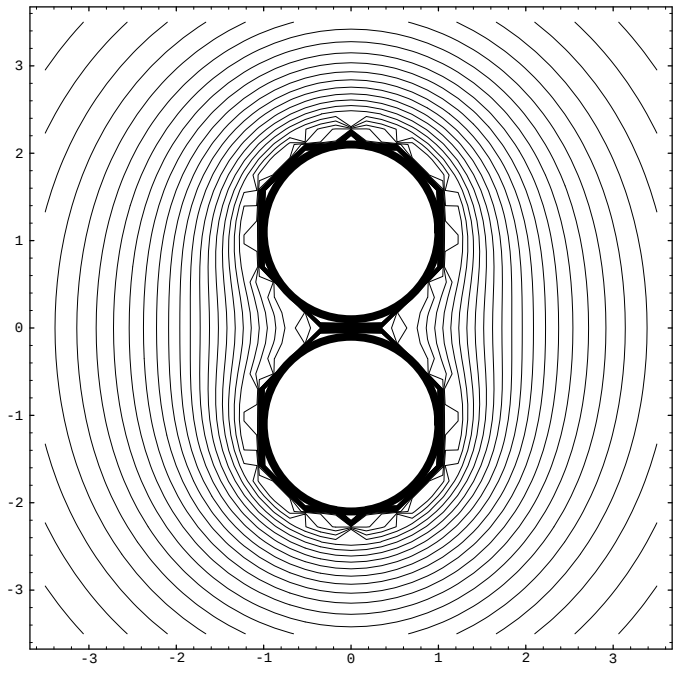

Figure 7.11: Two spheres in an unbounded domain.

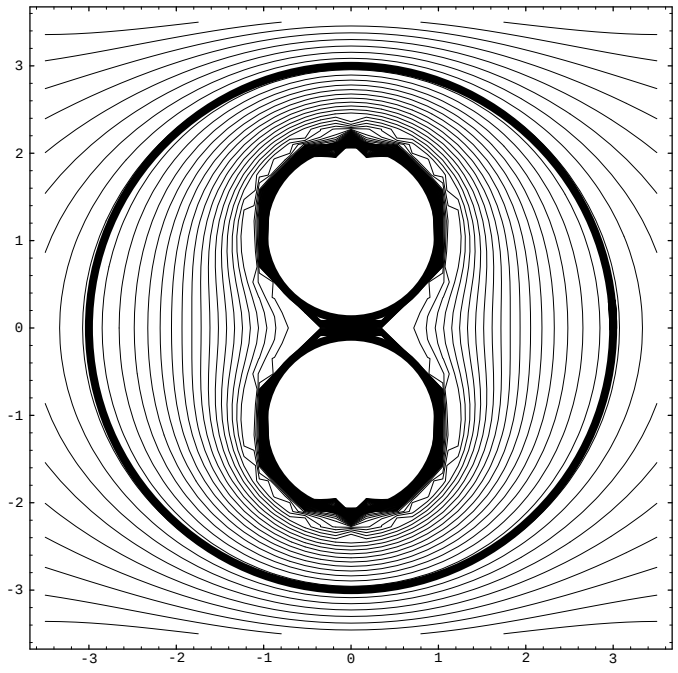

Figure 7.12: Two spheres inside a spherical container.

radius of container	3.0
potential on container's surface	0.0
center of container	$(0.0, 0.0, 0.0)$
radii of included spheres	1.0
net charge of one included sphere	1.0
center of 1. sphere	$(0.0, 0.0, -1.1)$
center of 2. sphere	$(0.0, 0.0, 1.1)$
number of BEs per sphere	320
no ambient electric field	
normal vector of planar crossection	$(0.0, 1.0, 0.0)$

Table 7.13: Input data for "Two spheres inside a spherical container".

radius of container	30
potential on container's surface	0.0
center of container	$(0.0, 0.0, 0.0)$
radii of included spheres	1.0 to 2.0
net charge of one included sphere	1.0
number of BEs per sphere	20
no ambient electric field	
normal vector of planar crossection	$(0.0, 0.0, 1.0)$

Table 7.14: Input data for "64 spheres inside a spherical container".

that the cross sections of the spheres are circles and only 5 out of the 64 spheres are intersected by this cross section. However, the spheres which are close to this cross section influence the shown equipotential lines significantly. The object outside the container in the lower left corner of the figure is the image of one of the included spheres. The data for this figure is listed in Table 7.14.

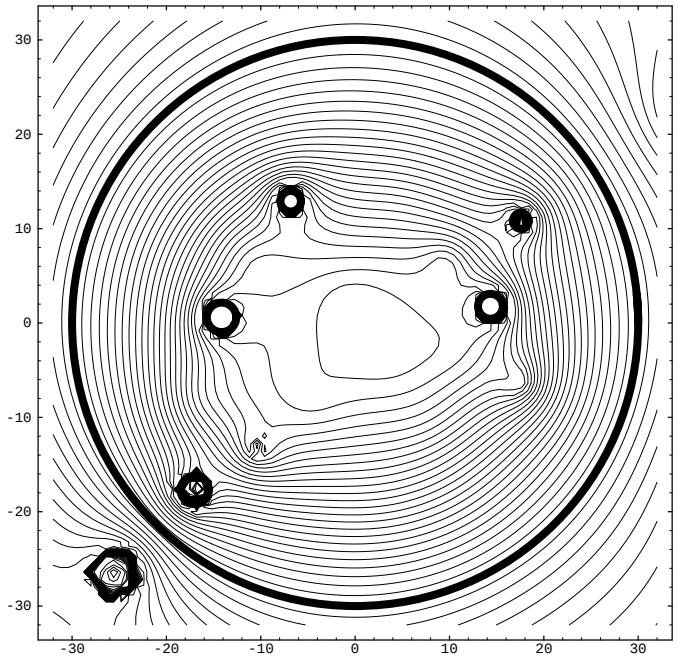

Figure 7.13: 64 spheres inside a spherical container.

number of spheres	32
radii of the spheres	0.25–0.75
distance between the spheres' centers	2.0
net charge of one sphere	0.0
superimposed electric field	1.0
domain: spherical container with radius	5.0
number of boundary elements per sphere	320
number of processors	32

Table 7.15: Input data for "32 spheres inside a spherical container".

7.5.2 Dielectric Constant and Thermal Conductivity

In order to verify the method calculating the dielectric constant and the thermal conductivity as described in Section 6, we placed 32 spheres as inclusions into a spherical container with a small volume fraction c. Analytical results were obtained by Zuzovsky and Brenner [53] for a two-phase material with a regular distribution of spherical inclusions in an unbounded 3-dimensional domain. The matrix surrounding the spheres is a vacuum and has the dielectric constant 1, the spheres are perfect conductors and have infinite dielectric constants. The 32 spheres are regularly distributed in a simple cubic array, as listed in Table 7.15 and shown in Figure 7.14.

Zuzovksy's and Brenner's formula for the effective dielectric constant for a regular distribution of spheres in a simple cubic lattice is:

$$\epsilon = 1 + 3c \left[1 - c - \frac{1.306 c^{10/3}}{1 - 0.407 c^{7/3}} - 0.022 c^{14/3} + O(c^6) \right]^{-1} . \tag{7.2}$$

There are three runs with different sphere sizes and volume fractions c as listed in Table 7.16,

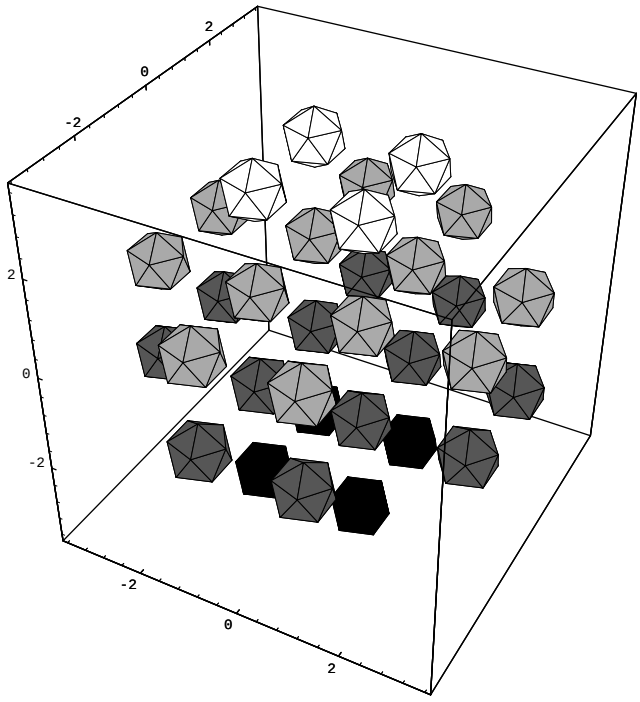

Figure 7.14: 32 spheres inside a spherical container, regular distribution in a simple cubic array. The graylevels represent the electric potential on their surfaces. Light-gray corresponds to low potential and dark-gray corresponds to high potential. The container with radius is not shown.

sphere radii	c	CDLBIEM	Zuzovsky and Brenner	relative error
0.25	0.004	1.01178	1.01205	0.00027
0.5	0.032	1.09697	1.09917	0.00020
0.75	0.108	1.35551	1.36355	0.00590

Table 7.16: Dielectric constant ϵ for different volume fractions c. Comparison of the results from CDLBIEM to the analytical results from Zuzovsky and Brenner.

which compares our results to the analytical results in Equation (7.2). Both results agree with a maximal relative error of 0.6% and propose CDLBIEM as a numerical method to investigate transport properties of two-phase materials.

Chapter 8

Summary and Conclusions

The Completed Double Layer Integral Equation Method (CDLBIEM) has been applied to the Laplace equation, in order to solve for the electric double layer density on N perfect conductors distributed in a 3-dimensional domain. The input data for CDLBIEM are the geometrical data and the net charges of the bodies and the superimposed electric field. The result for the double layer density returned by CDLBIEM has been used to calculate the electric potential on the bodies' surfaces and in the medium surrounding the bodies.

CDLBIEM has been implemented in three programming models on parallel machines: message-passing, static shared memory, and cooperative shared memory. The different aspects for writing our programs in these programming models have been discussed, and their performance has been analyzed.

CDLBIEM yields a linear algebraic equation system with a system matrix which has a small spectral radius. For this reason, this equation system has been solved very efficiently by Jacobi

iteration. The Jacobi iteration has been parallelized by distributing the N bodies onto the P processors. Each processor runs the Jacobi iteration simultaneously and asynchronously. The communication between the processors has been implemented either in the form of a request-and-send scheme or in the form of shared memory access.

This communication has been improved by implementing a communication schedule which naturally corresponds to the degree of physical interaction between the bodies. It reduces the amount of communicaton and computation since the interactions between distant bodies are not updated in every iteration step. Since the eigenvalues of the system matrix are dominated by the interactions between close bodies, this modified iteration still converges to the same solution, but about three times faster.

In order to solve for the double layer density on N bodies as inclusions in a spherical perfect conductor, CDLBIEM has been extended by using the method of images which yields zero potential on the surface of the spherical conductor. The resulting boundary integral equations for the double layer density have been solved by the same programs, but the routine which sets up the equations had to be modified. However, since the number of unknowns stays the same, there is neither more communication nor more memory usage than in the original case of an unbounded domain. The solution for the double layer density has been used to calculate the macroscopic dielectric constants and thermal conductivities of two-phase material which is surrounded by the spherical conductor.

CDLBIEM is a very efficient numerical method with a high accuracy. For instance, we have shown that its relative error is less than 0.5% for two almost touching spheres. The relative

error of the macroscopic dielectric constant is less than 0.6% compared to the result obtained by Zuzovsky and Brenner [53].

The comparison between the different implementations has shown that the cooperative shared memory model is the most promising of the three programming models for a standard programming model, which will allow the application programmer to easily port his or her programs from one computer to another. The program in the CSM model shows a 7.7-fold speedup from 16 to 128 processors and its time-constraint scaleup is almost perfect. The major advantage of the CSM model is its separation of the performance and the functionality of the communication. It provides a cost model for application programmers to optimize their algorithms and for computer architects to build parallel computers. Furthermore, its uniform address space provides transparent object names and easy programming.

Implementing programs on the message-passing model or in Split-C is more difficult, since the functionality and the performance of the communication are closely coupled. In the CMMD version the request-send-scheme had to be implemented in the form of message loops and in the Split-C version we had to insert low-level polling function calls in order to get good performance. In our case, Split-C is the better choice because of its very efficient implementation of point-to-point messages, in the form of active messages.

Virtual prototyping is a very good concept for developing parallel machines. On the Wisconsin Wind Tunnel it is easy to change system parameters, such as network latency, cache sizes, partition sizes, and to vary the algorithm and the computer architecture, in order to optimize the combination of both.

A widely accepted high-level programming model, for example CSM, will have a similar impact on parallel computing, as Fortran had in the early days of electronic digital computing. More application programmers, especially in the chemical engineering community, will be able to solve their computational expensive problems on high-performance parallel machines.

8.1 Future Work

The algorithm has been very efficiently implemented and executed in the CSM model. We have been able to run this program on a wide range of partition sizes and have obtained scaling results. However, we have not been able to obtain scaling results for the program in Split-C or the message-passing versions. In order to compare the performance of the three versions, we also need results on partitions larger than 32 processors for the Split-C and the message-passing versions.

So far, our program has solved problems with spheres. For these problems, linear planar triangles as boundary elements and one collocation point for the double layer density on one boundary element have been sufficient. Curved boundary elements, a collocation of higher order, and quadratures of higher order might be necessary to solve problems with bodies of more complex shapes.

The programs can be easily modified to solve the Stokes equations, the Navier equation, or the Stokes equations and the Laplace equation simultaneously (in order to simulate electrorheological fluids).

The behavior of CDLBIEM on the three discussed programming models and on other pro-

posed programming models has to be investigated in more detail. Our results should then affect the architecture of parallel computers.

Appendix A

Units in Electrostatics

All values of the electrostatic quantities discussed in this work are given in electrostatic units, esu. This appendix describes the conversion of electrostatic units to rationalized MKSA units or SI units.

Table A.1 lists some electrostatic quantities, their elecrostatic units, and their corresponding MKSA units according to [25].

Quantity	**Symbol**	**MKSA**	**esu**
Charge	Q	1C = 1As	$10c$ statcoulombs
Charge density	ρ	1C m^{-3}	$10^{-5}c$ statcoulombs cm^{-3}
Potential	ϕ	1V	$10^{6}c^{-1}$ statvolt
Electric field	$\boldsymbol{E}$	1V m^{-1}	$10^{4}c^{-1}$statvolt cm^{-1}

Table A.1: Electrostatic units and MKSA units. c is the speed of light, $c = 2.998 \cdot 10^{8}$m s^{-1} = $2.998 \cdot 10^{8}$.

Appendix B

CSM Program

In this appendix we give a tutorial on how to compile and run the CSM program on the WWT. The source code, the make file, the shell scripts, and templates for the input data are listed. Please refer to "A Programming Tutorial (Or How to Survive Your First Few Days on the Wisconsin Wind Tunnel)" by Alain Kaegi and Shubhendu S. Mukherjee (Computer Sciences Department, University of Wisconsin – Madison) for a general introduction into the WWT. The programming structure of the WWT may change, so please contact the people who develop the WWT for up-to-date information. Please contact me if you are interested in a diskette with the source files for the message-passing, the Split-C, or the CSM programs. My Internet address is: `traenkle@luther.che.wisc.edu`.

B.1 Benchmark Tree and Source Files

First of all, a benchmark tree has to be created in any directory accessible by the CM-5. This benchmark tree is a hierarchical structure of subdirectories and contains the source codes, the make files, the shell scripts, and the input and output files of several benchmarks. A benchmark is a program for testing the performance of a computer. For example, our program is a benchmark. The benchmark tree as shown in Figure B.1 has to be created by hand using the UNIX command `mkdir`. The arrows are links which are created by `ln -S`. All the bold names are directory names, the other names are file names. The files listed in this appendix have to be placed into the locations according to Figure B.1. The name of the benchmark root directory can be chosen arbitrarily, we chose `BM`. The other names are fixed.

The source code for the benchmark `lap`, which is our CSM code, consists of two modules:

`lap.U`	CSM source code
`geometry.c`	C source code

Both modules include the C header files:

`my.h`	C header file
`geometry.h`	C header file

`lap.mak.include` is a general make file which is included by the make file `Makefile`. `lap.mak.include` should not be changed, whereas `Makefile` should be changed to set C macros or to choose between different cache coherence protocols. The C macros supported by `lap.U` are:

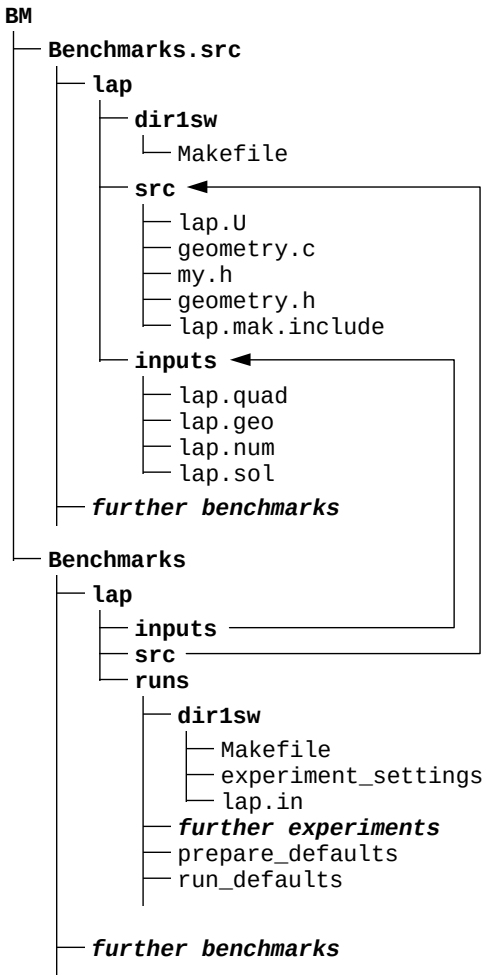

Figure B.1: Benchmark tree. Bold names are directories, other names are files. The arrows are UNIX links.

`WWT`	The WWT related source lines are activated. Note that `lap.U` can also be used as a sequential code.
`CICO`	`check_out_S` is used to check out the remote solution subvectors.
`CICO_X`	`check_out_X` is used to check out the remote solution subvectors.
`LOCK_`	The remote solution subvectors are locked while accessing them.
`TIMERS`	The virtual timers of WWT are activated.
`CONTAINER`	The linear algebraic equation system is set up, in order to model N bodies in a container.
`SINGLEPREC`	Single-precision is used for floating-point numbers.

`experiment_settings` (Figure B.2) is a shell script which sets the run-time arguments. The default run-time arguments are defined in `run_defaults` (Figure B.3). `prepare_defaults` defines the default way how to create new directories for experiments (runs).

B.2 Compiling the Executable

The following C-shell environmental variables have to be set or appended by the UNIX commands `setenv` or `set` respectively:

```
WWT_ROOT   /p/WWT
BMTREE     BM
path       (/p/WWT/Bin /p/WWT/Scripts)
```

Then, the executable `lap.Z` can be created by running the command `wwt_build -t$BMTREE -blap dir1sw`. The compressed executable `lap.Z` is written to the directory `BM/Benchmarks/lap/runs/dir1sw`.

B.3 Input and Output Files

The main input file `lap.in` contains the names of all other input and output files. Each row contains a file name and controls whether an output file is created. If a row for an output file name is left blank, the corresponding output file is not created and its data is not calculated. A sample for `lap.in` is in Figure B.4. The rows have the following functions:

1.	`240`	240/256 of the system matrix is stored in memory
2.	`../../inputs/lap.quad`	input file for quadrature points and weights
3.	`../../inputs/lap.num`	input file for numerical data
4.	`../../inputs/lap.geo`	input file for geometrical data
5.	`lap.psi`	output file for potential on spheres' surfaces
6.	`lap.phi`	output file for double layer density on boundary elements
7.	`lap.err`	output file for relative error
7.	`../../inputs/lap.sol`	this error is based on the accurate solution in this input file
8.	`lap.time`	output file for virtual timers
9.	`loadbalance`	switches the load balancing routine on or off

Templates for the other input files are given in Figure B.5 for `lap.quad`, in Figure B.6 for `lap.num`, in Figure B.7 for `lap.geo`, and in Figure B.8 for `lap.sol`.

B.4 Running the Executable

By calling `wwt_run -nowait -t$BMTREE -blap dir1sw`, a job is submitted to the job manager of the CM-5. The output files are written to the directory `BM/Benchmarks/lap/runs/dir1sw`.

Note that new directories for different runs can be created in `BM/Benchmarks/lap/runs`, for example for different input sets, different program versions, or different cache coherence

protocols. By setting C macros in the different make files, the same source code `lap.U` can be used for all cases. `wwt_build` creates different executables for different runs and places them into the corresponding directories.

```
setenv EXP_FLAGS "-NPT -RO1 -n 64 -nl 100 -ql 100"
setenv CMD_ARGS "lap.in"
setenv JSUB_CPU "-cputime 60min"
setenv JSUB_MEM "-memory 300mb"
```

Figure B.2: Shell script experiment_settings.

```
#!/usr/misc/tcsh -f
#
# Defaults values for the various experiment parameters.
#
# Don't make any change to this file, if you need different values
# for the variables below, define them in the script calling
# "run".
#
# EXPERIMENT has no longer a default, this way once can detect
# unintentional use of ./run.
#
# CMD_IN is defined only if input is to be read from stdin.
#

# if ( ! $?EXPERIMENT ) setenv EXPERIMENT "dir1sw"
if ( ! $?SIM_NAME ) setenv SIM_NAME "{$WWT_ROOT}/Bin/dir1sw"
if ( ! $?SIM_ARGS ) setenv SIM_ARGS ""
if ( ! $?CMD_NAME ) setenv CMD_NAME "lap"
if ( ! $?CMD_ARGS ) setenv CMD_ARGS "../../inputs/lap.in.64.20.64.d1"
#if ( ! $?CMD_IN )
if ( ! $?JSUB_NPROC ) setenv JSUB_NPROC "-nproc 32"
if ( ! $?JSUB_CPU ) setenv JSUB_CPU "-cputime 30min"
if ( ! $?JSUB_MEM ) setenv JSUB_MEM "-memory 300mb"
if ( ! $?JSUB_ME ) setenv JSUB_ME "-mail_end"
if ( ! $?JSUB_EXP ) setenv JSUB_EXP "-export"
if ( ! $?JSUB_XTRA ) setenv JSUB_XTRA "-server mendota"
```

Figure B.3: Shell script run_defaults.

```
240
../../inputs/lap.quad
../../inputs/lap.num
../../inputs/lap.geo
lap.psi
lap.sol
lap.err ../../inputs/lap.sol
lap.time
loadbalance
```

Figure B.4: Main input file lap.in.

```
(number of quadrature points Q)
Q rows of (quadrature point,  corresponding weight, separated by a space)
```

Figure B.5: Input file lap.quad.

```
(number of spheres N)
(number of elements of initial polyhedron)
(number of tesselation steps to create final polyhedron)
(number of iteration steps)
(CS matrix) (N*N entries, separated by spaces or carriage returns)
```

Figure B.6: Input file lap.num.

```
(number of spheres N)
(superimposed electric field, 3 entries, separated by spaces)
N rows of (x, y, z coordinates, radius, net charge) for each sphere
```

Figure B.7: Input file lap.num.

```
N * M rows of (double layer density on boundary element)
```

Figure B.8: Input file lap.sol.

B.5 Listings

B.5.1 Module lap.U

```
/*------------------------------------------------------------
   lap.U / lap.c
   5.0
  ------------------------------------------------------------
   used modules:
   geometry.c
  ------------------------------------------------------------
   programmed by Frank Traenkle
   last modification: 11/20/93 19:41
  ------------------------------------------------------------
   WWT version and serial version combined
   Ready for Container Problem
   each processor has subset of rows
   buffered submatrices
   N particles in 3D-space
   advanced output
  ------------------------------------------------------------
   shared and global data
  ------------------------------------------------------------*/

/*
   Deflation Version, solved by Jacobi Iterations
   No Gaussian Quadratures if elements are not near neighbors.
*/


/*----------------------------
   DEFINEs
  ----------------------------*/
/* serial version */
#ifndef WWT
#  define XX_NUM_NODES 1
#  define G_MALLOC(ulSize) malloc(ulSize);
#endif

#ifdef CICO_X
#  define CICO
#endif

#define FILENAM_SIZE 80
#define STRING_SIZE 80

/* virtual timers */
#define VT_SOLVE 0
```

```
#define VT_SOL_READ 1
#define VT_SOL_WRITE 2
#define VT_RECREATEMATRICES 3
#define VT_DOTPRODUCT 4
#define VT_CALCPSI 5
#define VT_CALCERR 6
#define VT_CREATEMATRICES 7
#define VT_WORK 8
#define VT_ 9             /* maximal number of virtual timers */

/* virtual timer macros */
#ifdef TIMERS
#  define VTTYPE unsigned
#  define VTSTART();                                        \
     avtStart[ulVT++] = wwt_get_vt_lo();
#  define VTSTOP(ulVTSym);   \
     avt[ulVTSym] += wwt_get_vt_lo() - avtStart[--ulVT];
/*
#  define VTSTART();                                        \
     avtStart[ulVT++] = wwt_get_vt_dbl();
#  define VTSTOP(ulVTSym);   \
     avt[ulVTSym] += wwt_get_vt_dbl() - avtStart[--ulVT];
*/
#else
#  define VTSTART(); ;
#  define VTSTOP(ulVTSym); ;
#endif

#ifdef WWT
#  define PRINT_ADDRESS(sz, p) \
    fprintf(pFileStd, "%20s %12u %12u\n", sz, p, (ULONG)p % (ULONG)BLOCK_SIZE);
#endif

#ifdef CONTAINER
#  define CONTAINER_1 \
        fpRy = AABS(afpY);\
        afpD[0] = afpY[0] / fpRy;\
        afpD[1] = afpY[1] / fpRy;\
        afpD[2] = afpY[2] / fpRy;\
        fpRz = pg->fpR2 / fpRy;\
        afpZ[0] = afpD[0] * fpRz;\
        afpZ[1] = afpD[1] * fpRz;\
        afpZ[2] = afpD[2] * fpRz;\
        afpXmZ[0] = afpX[0] - afpZ[0];\
        afpXmZ[1] = afpX[1] - afpZ[1];\
        afpXmZ[2] = afpX[2] - afpZ[2];\
        fpRxz = AABS(afpXmZ);\
        for (ulII = 0; ulII < 3; ulII++)\
        {\
          afpSum1[ulII] = 0.;\
          for (ulJJ = 0; ulJJ < 3; ulJJ++)\
```

```
          {\
            afpSum1[ulII] += (DELTA(ulII, ulJJ) - 2. * afpD[ulII] *\
              afpD[ulJJ]) * afpXmZ[ulJJ];\
          }\
        }
#define CONTAINER_2 \
          ((pgeoJ->afpN[0]*afpXmY[0] +\
            pgeoJ->afpN[1]*afpXmY[1] +\
            pgeoJ->afpN[2]*afpXmY[2]) /\
           (fpRxy * fpRxy * fpRxy) +\
           (-fpRz *\
            (pgeoJ->afpN[0] * afpSum1[0] +\
             pgeoJ->afpN[1] * afpSum1[1] +\
             pgeoJ->afpN[2] * afpSum1[2]) /\
            SQR(fpRxz) +\
            pgeoJ->afpN[0] * afpD[0] +\
            pgeoJ->afpN[1] * afpD[1] +\
            pgeoJ->afpN[2] * afpD[2]) *\
           SQR(fpRz) / (fpRxz * pg->fpR3));
#endif

#ifdef WWT
/*----------------------------
   Initialize WWT Environment
  ----------------------------*/
ENV
#endif

/*----------------------------
   INCLUDEs
  ----------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <assert.h>
#ifdef WWT
#include <align.h>
#endif
#include "my.h"
#include "geometry.h"

/*----------------------------
   TYPEDEFs
  ----------------------------*/
/* body data */
typedef struct _BODY
{
  ULONG ulNumber; /* original number */
  FPTYPE afpZ[3]; /* center */
  FPTYPE fpR;     /* radius */
```

```
  FPTYPE fpQ;     /* charge */
} BODY, *PBODY;

#ifdef WWT
/* lock declaration */
typedef LOCKDEC(XLOCK)
#endif
/* global data space */
typedef struct _GLOBAL
{
  /* --- R/O data during calculation --- */
  /* Gaussian-Legendre Quadrature */
  USHORT usQuadN;
  FPTYPE *afpQuadX;
  FPTYPE *afpQuadW;
  FPTYPE *afpQuadXt;
  FPTYPE *afpQuadWt;
  /* Stored Element Data */
  PGEO ageo;
  FPTYPE fpGeoA;
#ifdef CONTAINER
  /* radius of spherical container */
  FPTYPE fpR;
  /* its square, cube */
  FPTYPE fpR2;
  FPTYPE fpR3;
#endif
  /* Superimposed Electric Field */
  FPTYPE afpE0[3];
  /* Geometrical Data of Particles */
  PBODY abody;
  /* --- elements in iteration */
  /* numbers */
  ULONG ulN;
  ULONG ulM;
  ULONG ulNM;
  ULONG ulNP;
  ULONG ulNMP;
  ULONG ulNBuf;
  /* output mode */
  USHORT fOutMode;         /* flag word for output modes
                              0 0 0 0  (bit mask)
                              | | | |
                              | | | -----> output psi
                              | | -------> output final solution
                              | ---------> output error
                              -----------> output timers
                            */
  /* number of iteration steps */
  ULONG ulIterN;
  /* priority matrix */
```

```
  ULONG *aulBufPrio;
  /* exact solution vector */
  FPTYPE *afpXSol;
  /* --- elements in initialization, output --- */
#ifdef WWT
  /* processor number (unique) */
  ULONG ulProc;
  /* lock on node number */
  XLOCK xulProc;
#endif
  /* Numerical Data */
  USHORT usPolyN;
  USHORT usTessN;
#ifdef WWT
  /* --- dummy variable to align next cache block to 32 bytes --- */
#  ifdef CONTAINER
#    ifdef SINGLEPREC
  ULONG aulDummy[5];
#    else
  ULONG aulDummy[6];
#    endif
#  else
#    ifndef SINGLEPREC
  ULONG aulDummy[4];
#    endif
#  endif
#endif
  /* --- read and write data --- */
  /* whole solution vector of previous iteration */
  FPTYPE *afpXOld;
#ifdef WWT
  /* locks for different parts of this vector */
  XLOCK *axafpXOld;
#endif
  /* surface potential */
  FPTYPE *afpPsi;
  /* error in each iteration */
  FPTYPE *afpXErr;
#ifdef WWT
  /* lock this vector */
  XLOCK xafpXErr;
#endif
  /* virtual timers */
#ifdef TIMERS
  VTTYPE (*avt)[VT_];
#endif
} GLOBAL, *PGLOBAL;

/*----------------------------
   CONSTANTs
  ----------------------------*/
```

```
/*----------------------------
   VARIABLEs
  ----------------------------*/
/* standard output file */
static FILE *pFileStd;
/* output files */
static char szFileOutPsi[FILENAM_SIZE];
static char szFileOutSol[FILENAM_SIZE];
static char szFileOutErr[FILENAM_SIZE];
static char szFileOutTime[FILENAM_SIZE];
/* input files */
static char szFileIn[FILENAM_SIZE];
static char szFileInSol[FILENAM_SIZE];
static char szFileQuad[FILENAM_SIZE];
static char szFileNum[FILENAM_SIZE];
static char szFileGeo[FILENAM_SIZE];
/* my processor number */
static ULONG ulProc;
/* range of rows on processor */
static ULONG ulRowStart, ulRowStop;
/* range of particles on processor */
static ULONG ulParticleStart, ulParticleStop;
/* reference to buffered submatrices in afpA */
static ULONG *aulABufRef;
/* system matrix A */
static FPTYPE *afpA;
/* RHS vector b */
static FPTYPE *afpB;
/* virtual timers */
#ifdef TIMERS
static VTTYPE avtStart[VT_];  /* array for start times */
static ULONG ulVT;              /* current timer */
static VTTYPE avt[VT_];       /* times on current node */
#endif
/* load balance flag */
static BOOL fLoadBalance;

/* global data space */
PGLOBAL pg;

/*----------------------------
   PROTOTYPEs
  ----------------------------*/
static void Initialize();
static void StoreElements();
static void AdjustGeometry();
static void LoadBalance();
static void OutputInfo();
static void OutputPsi();
static void OutputSol();
```

```
static void OutputErr();
#ifdef TIMERS
static void OutputVT();
static void OutputVT1();
static void VTMinMax();
#endif
void Work();
static void InitializeWork();
static void ExitWork();
static void CreateMatrices();
static void CreateAnn();
static void CreateAkl();
static void CreateBn();
static void Solve();

/*----------------------------
   MAIN
  ----------------------------*/
int main(iAN, szA)
int iAN;
char *szA[];
{
#ifdef WWT
  /* init WWT environment */
  INITENV
#endif

  /* get general input file name from argument list */
  assert(iAN == 2);
  strcpy(szFileIn, szA[1]);

  /* open standard output file */
/*
  pFileStd = fopen("out", "w");
*/
  pFileStd = stdout;

  fprintf(pFileStd, "--- Initialize ---\n");
  Initialize();
  fprintf(pFileStd, "--- CreateGeometry ---\n");
  CreateGeometry(pg->usPolyN, pg->usTessN);
  fprintf(pFileStd, "--- StoreElements ---\n");
  StoreElements();
  fprintf(pFileStd, "--- AdjustGeometry ---\n");
  AdjustGeometry();
  fprintf(pFileStd, "--- CloseGeometry ---\n");
  CloseGeometry();
  if (fLoadBalance)
  {
    printf("--- LoadBalance ---\n");
    LoadBalance();
```

```
  }
  /* start parallel code */
  fprintf(pFileStd, "--- CREATE_ALL ---\n");
#ifdef WWT
  CREATE_ALL(Work)
#endif
  Work();
#ifdef WWT
  /* synchronize nodes */
  WAIT_FOR_END()
#endif
  /* do output */
  if (pg->fOutMode & 1)
  {
    fprintf(pFileStd, "--- OutputPsi ---\n");
    OutputPsi();
  }
  if (pg->fOutMode & 2)
  {
    fprintf(pFileStd, "--- OutputSol ---\n");
    OutputSol();
  }
  if (pg->fOutMode & 4)
  {
    fprintf(pFileStd, "--- OutputErr ---\n");
    OutputErr();
  }
#ifdef TIMERS
  if (pg->fOutMode & 8)
  {
    fprintf(pFileStd, "--- OutputVT ---\n");
    OutputVT();
  }
#endif
  return 0;
}

/*----------------------------
   INITIALIZE
  ----------------------------*/
static void Initialize()
{
  char sz[FILENAM_SIZE * 3];
  FILE *pFileIn;
  FILE *pFileQuad;
  FILE *pFileNum;
  FILE *pFileGeo;
  FILE *pFileInSol;
  USHORT usI, usJ;
  FPTYPE fpXtMax;
  ULONG ulI;
```

```
  ULONG *pul;
  ULONG ulN;

  /* allocate global data space */
  pg = (PGLOBAL)G_MALLOC(sizeof(GLOBAL))
  assert(pg != NULL);

#ifdef WWT
  /* initialize processor number */
  pg->ulProc = 0;
#endif

  /* at first no output mode */
  pg->fOutMode = 0;
  /* open general input file */
  pFileIn = fopen(szFileIn, "r");
  assert(pFileIn != NULL);
  /* number of buffered sub matrices per particle row */
  fgets(sz, sizeof(sz), pFileIn);
  pg->ulNBuf = (ULONG)atol(sz);
  /* get file names */
  fgets(szFileQuad, sizeof(szFileQuad), pFileIn);
  szFileQuad[strlen(szFileQuad) - 1] = '\0';
  fgets(szFileNum, sizeof(szFileNum), pFileIn);
  szFileNum[strlen(szFileNum) - 1] = '\0';
  fgets(szFileGeo, sizeof(szFileGeo), pFileIn);
  szFileGeo[strlen(szFileGeo) - 1] = '\0';
  fgets(szFileOutPsi, sizeof(szFileOutPsi), pFileIn);
  szFileOutPsi[strlen(szFileOutPsi) - 1] = '\0';
  if (strlen(szFileOutPsi) > 0)
  {
    /* output surface potentials for each iteration */
    pg->fOutMode |= 1;
  }
  fgets(szFileOutSol, sizeof(szFileOutSol), pFileIn);
  szFileOutSol[strlen(szFileOutSol) - 1] = '\0';
  if (strlen(szFileOutSol) > 0)
  {
    /* output final phi vector */
    pg->fOutMode |= 2;
  }
  fgets(sz, sizeof(sz), pFileIn);
  sz[strlen(sz) - 1] = '\0';
  if (strlen(sz) > 0)
  {
    /* output error for each iteration */
    pg->fOutMode |= 4;
    sscanf(sz, "%s %s", szFileOutErr, szFileInSol);
  }
  fgets(szFileOutTime, sizeof(szFileOutTime), pFileIn);
  szFileOutTime[strlen(szFileOutTime) - 1] = '\0';
```

```
  if (strlen(szFileOutTime) > 0)
  {
    /* output times */
    pg->fOutMode |= 8;
  }
  /* Load Balance ? */
  fgets(sz, sizeof(sz), pFileIn);
  sz[strlen(sz) - 1] = '\0';
  if (strlen(sz) > 0)
  {
    fLoadBalance = 1;
  }
  else
  {
    fLoadBalance = 0;
  }
  /* close general input file */
  fclose(pFileIn);

/*
  printf("%s\n", szFileQuad);
  printf("%s\n", szFileNum);
  printf("%s\n", szFileGeo);
*/

  /* read input file for Gaussian-Legendre Quadrature parameters */
  pFileQuad = fopen(szFileQuad, "r");
  assert(pFileQuad != NULL);
  fscanf(pFileQuad, "%hu", &pg->usQuadN);
  /* allocate memory for data */
  pg->afpQuadX = (FPTYPE *)G_MALLOC(pg->usQuadN * sizeof(FPTYPE))
  assert(pg->afpQuadX != NULL);
  pg->afpQuadW = (FPTYPE *)G_MALLOC(pg->usQuadN * sizeof(FPTYPE))
  assert(pg->afpQuadW != NULL);
  for (usI = 0; usI < pg->usQuadN; usI++)
  {
#ifdef SINGLEPREC
    fscanf(pFileQuad, "%f %f", &pg->afpQuadX[usI], &pg->afpQuadW[usI]);
#else
    fscanf(pFileQuad, "%lf %lf", &pg->afpQuadX[usI], &pg->afpQuadW[usI]);
#endif
  }
  fclose(pFileQuad);
  /* calculate scaled parameters */
  pg->afpQuadXt = (FPTYPE *)G_MALLOC(pg->usQuadN * pg->usQuadN * sizeof(FPTYPE))
  assert(pg->afpQuadXt != NULL);
  pg->afpQuadWt = (FPTYPE *)G_MALLOC(pg->usQuadN * pg->usQuadN * sizeof(FPTYPE))
  assert(pg->afpQuadWt != NULL);
  for (usI = 0; usI < pg->usQuadN; usI++)
  {
    fpXtMax = 1. - pg->afpQuadX[usI];
```

```
    for (usJ = 0; usJ < pg->usQuadN; usJ++)
    {
      *(pg->afpQuadXt+usI*pg->usQuadN+usJ) = pg->afpQuadX[usJ] * fpXtMax;
      *(pg->afpQuadWt+usI*pg->usQuadN+usJ) = pg->afpQuadW[usJ] * fpXtMax;
    }
  }

  /* read input file for numerical data */
  pFileNum = fopen(szFileNum, "r");
  assert(pFileNum != NULL);
  fscanf(pFileNum, "%u", &pg->ulN);
  /* pg->ulN must be multiple of XX_NUM_NODES */
  assert(pg->ulN % (ULONG)XX_NUM_NODES == 0);
  /* pg->ulN must be greater or equal than pg->ulNBuf */
  assert(pg->ulN >= pg->ulNBuf);
  fscanf(pFileNum, "%hu", &pg->usPolyN);
  fscanf(pFileNum, "%hu", &pg->usTessN);
  fscanf(pFileNum, "%u", &pg->ulIterN);
  /* read priority matrix */
  pg->aulBufPrio = (ULONG *)G_MALLOC(pg->ulN * pg->ulN * sizeof(ULONG))
  assert(pg->aulBufPrio != NULL);
  for (ulI = 0, pul = pg->aulBufPrio;
       ulI < pg->ulN * pg->ulN;
       ulI++, pul++)
  {
    fscanf(pFileNum, "%u", pul);
  }
  fclose(pFileNum);

  /* loop boundaries */
  pg->ulM = pg->usPolyN;
  for (usI = 0; usI < pg->usTessN; usI++)
  {
    pg->ulM *= 4;
  }
  pg->ulNM = pg->ulN * pg->ulM;
  pg->ulNP = pg->ulN / XX_NUM_NODES;
  pg->ulNMP = pg->ulNM / XX_NUM_NODES;
  /* allocate memory */
  pg->ageo = (PGEO)G_MALLOC(pg->ulM * sizeof(GEO))
  assert(pg->ageo != NULL);
  pg->abody = (PBODY)G_MALLOC(pg->ulN * sizeof(BODY))
  assert(pg->abody != NULL);
  pg->afpXOld = (FPTYPE *)G_MALLOC(pg->ulNM * sizeof(FPTYPE))
  assert(pg->afpXOld != NULL);
#ifdef WWT
  pg->axafpXOld = (XLOCK *)G_MALLOC(pg->ulN * sizeof(XLOCK))
  assert(pg->axafpXOld != NULL);
#endif
#ifdef TIMERS
  pg->avt =
```

```
    (VTTYPE (*)[VT_])G_MALLOC(XX_NUM_NODES * VT_ * sizeof(VTTYPE))
  assert(pg->avt != NULL);
#endif
#ifdef WWT
  /* initialize locks */
  LOCKINIT(pg->xulProc)
  for (ulI = 0; ulI < pg->ulN; ulI++)
  {
    LOCKINIT(pg->axafpXOld[ulI])
  }
  LOCKINIT(pg->xafpXErr)
#endif
  /* Superimposed Electric Field */
  /* if output of psi */
  if (pg->fOutMode & 1)
  {
    /* allocate memory for surface potential */
    pg->afpPsi = (FPTYPE *)G_MALLOC(pg->ulIterN * pg->ulN * sizeof(FPTYPE))
    assert(pg->afpPsi != NULL);
  }
  /* if output of error */
  if (pg->fOutMode & 4)
  {
    /* allocate memory for error */
    pg->afpXErr = (FPTYPE *)G_MALLOC(pg->ulIterN * sizeof(FPTYPE))
    assert(pg->afpXErr != NULL);
    /* initialize it with 0s */
    for (ulI = 0; ulI < pg->ulIterN; ulI++)
    {
      pg->afpXErr[ulI] = 0.;
    }
    /* allocate memory for exact solution vector */
    pg->afpXSol = (FPTYPE *)G_MALLOC(pg->ulNM * sizeof(FPTYPE))
    assert(pg->afpXSol != NULL);
    /* read it from file */
    pFileInSol = fopen(szFileInSol, "r");
    for (ulI = 0; ulI < pg->ulNM; ulI++)
    {
#ifdef SINGLEPREC
      fscanf(pFileInSol, "%f", &pg->afpXSol[ulI]);
#else
      fscanf(pFileInSol, "%lf", &pg->afpXSol[ulI]);
#endif
    }
    fclose(pFileInSol);
  }

  /* read input file for geometrical data */
  pFileGeo = fopen(szFileGeo, "r");
  assert(pFileGeo != NULL);
  fscanf(pFileGeo, "%u", &ulN);
```

```
  /* check if geo file is consistent to num file */
  assert(pg->ulN == ulN);
#ifdef SINGLEPREC
  fscanf(pFileGeo, "%f %f %f",
         &pg->afpE0[0], &pg->afpE0[1], &pg->afpE0[2]);
#else
  fscanf(pFileGeo, "%lf %lf %lf",
         &pg->afpE0[0], &pg->afpE0[1], &pg->afpE0[2]);
#endif
  for (ulI = 0; ulI < pg->ulN; ulI++)
  {
#ifdef SINGLEPREC
    fscanf(pFileGeo, "%f %f %f %f %f",
           &pg->abody[ulI].afpZ[0], &pg->abody[ulI].afpZ[1],
           &pg->abody[ulI].afpZ[2],
           &pg->abody[ulI].fpR, &pg->abody[ulI].fpQ);
#else
    fscanf(pFileGeo, "%lf %lf %lf %lf %lf",
           &pg->abody[ulI].afpZ[0], &pg->abody[ulI].afpZ[1],
           &pg->abody[ulI].afpZ[2],
           &pg->abody[ulI].fpR, &pg->abody[ulI].fpQ);
#endif
    pg->abody[ulI].ulNumber = ulI;
  }
#ifdef CONTAINER
#  ifdef SINGLEPREC
  fscanf(pFileGeo, "%f", &pg->fpR);
#  else
  fscanf(pFileGeo, "%lf", &pg->fpR);
#  endif
  /* its square, cube */
  pg->fpR2 = SQR(pg->fpR);
  pg->fpR3 = pg->fpR2 * pg->fpR;
#endif

  fclose(pFileGeo);

#ifdef WWT
  PRINT_ADDRESS("usQuadN", &pg->usQuadN);
  PRINT_ADDRESS("afpQuadX", &pg->afpQuadX);
  PRINT_ADDRESS("afpQuadW", &pg->afpQuadW);
  PRINT_ADDRESS("afpQuadXt", &pg->afpQuadXt);
  PRINT_ADDRESS("afpQuadWt", &pg->afpQuadWt);
  PRINT_ADDRESS("ageo", &pg->ageo);
  PRINT_ADDRESS("fpGeoA", &pg->fpGeoA);
#ifdef CONTAINER
  PRINT_ADDRESS("fpR", &pg->fpR);
  PRINT_ADDRESS("fpR2", &pg->fpR2);
  PRINT_ADDRESS("fpR3", &pg->fpR3);
#endif
  PRINT_ADDRESS("afpE0", &pg->afpE0[0]);
```

```
  PRINT_ADDRESS("abody", &pg->abody);
  PRINT_ADDRESS("ulN", &pg->ulN);
  PRINT_ADDRESS("ulM", &pg->ulM);
  PRINT_ADDRESS("ulNM", &pg->ulNM);
  PRINT_ADDRESS("ulNP", &pg->ulNP);
  PRINT_ADDRESS("ulNMP", &pg->ulNMP);
  PRINT_ADDRESS("ulNBuf", &pg->ulNBuf);
  PRINT_ADDRESS("fOutMode", &pg->fOutMode);
  PRINT_ADDRESS("ulIterN", &pg->ulIterN);
  PRINT_ADDRESS("aulBufPrio", &pg->aulBufPrio);
  PRINT_ADDRESS("afpXSol", &pg->afpXSol);
  PRINT_ADDRESS("ulProc", &pg->ulProc);
  PRINT_ADDRESS("xulProc", &pg->xulProc);
  PRINT_ADDRESS("usPolyN", &pg->usPolyN);
  PRINT_ADDRESS("usTessN", &pg->usTessN);
#if !defined(SINGLEPREC) || defined(CONTAINER)
  PRINT_ADDRESS("aulDummy", &pg->aulDummy[0]);
#endif
  PRINT_ADDRESS("afpXOld", &pg->afpXOld);
  PRINT_ADDRESS("axafpXOld", &pg->axafpXOld);
  PRINT_ADDRESS("afpPsi", &pg->afpPsi);
  PRINT_ADDRESS("afpXErr", &pg->afpXErr);
  PRINT_ADDRESS("xafpXErr", &pg->xafpXErr);
#ifdef TIMERS
  PRINT_ADDRESS("avt", &pg->avt);
#endif
#endif
}

/*----------------------------
   STOREELEMENTS
  ----------------------------*/
static void StoreElements()
{
  ULONG ulI;

  pg->fpGeoA = 0.;
  for (ulI = 0; ulI < pg->ulM; ulI++)
  {
    GetElementAll(ulI, &pg->ageo[ulI]);
    pg->fpGeoA += pg->ageo[ulI].fpDA;
  }
}

/*----------------------------
   ADJUSTGEOMETRY
  ----------------------------*/
static void AdjustGeometry()
{
  ULONG ulI;
  FPTYPE fpRadiusPolyInv;
```

```
  PGEO pgeo;

  fpRadiusPolyInv = sqrt((4. * PI) / pg->fpGeoA);
  for (ulI = 0; ulI < pg->ulM; ulI++)
  {
    pgeo = &pg->ageo[ulI];
    pgeo->afpP1[0] *= fpRadiusPolyInv;
    pgeo->afpP1[1] *= fpRadiusPolyInv;
    pgeo->afpP1[2] *= fpRadiusPolyInv;
    pgeo->afpP2[0] *= fpRadiusPolyInv;
    pgeo->afpP2[1] *= fpRadiusPolyInv;
    pgeo->afpP2[2] *= fpRadiusPolyInv;
    pgeo->afpP3[0] *= fpRadiusPolyInv;
    pgeo->afpP3[1] *= fpRadiusPolyInv;
    pgeo->afpP3[2] *= fpRadiusPolyInv;
    pgeo->afpU1[0] *= fpRadiusPolyInv;
    pgeo->afpU1[1] *= fpRadiusPolyInv;
    pgeo->afpU1[2] *= fpRadiusPolyInv;
    pgeo->afpU2[0] *= fpRadiusPolyInv;
    pgeo->afpU2[1] *= fpRadiusPolyInv;
    pgeo->afpU2[2] *= fpRadiusPolyInv;
    pgeo->afpX[0] *= fpRadiusPolyInv;
    pgeo->afpX[1] *= fpRadiusPolyInv;
    pgeo->afpX[2] *= fpRadiusPolyInv;
    pgeo->fpDA *= fpRadiusPolyInv * fpRadiusPolyInv;
    pgeo->fpJacob *= fpRadiusPolyInv * fpRadiusPolyInv;
  }
  pg->fpGeoA *= fpRadiusPolyInv * fpRadiusPolyInv;
}

/*-------------------------------------
   LOADBALANCE
  -------------------------------------*/
static void LoadBalance()
{
  ULONG ulI, ulJ, ulK;
  ULONG ulP;
  ULONG *aulCrit;
  ULONG *aulCritSum;
  ULONG *aulN;
  ULONG *aulPrio;
  ULONG ulCritMax;
  ULONG ulIMax;
  ULONG ulCritSumMin;
  ULONG ulPMin;
  BODY bodyTemp;
  FPTYPE fpTemp;
  ULONG ulTemp;
  ULONG ulSum;
  ULONG ulMin, ulMax;
```

```
aulCrit = (ULONG *)malloc(pg->ulN * sizeof(ULONG));
assert(aulCrit != NULL);
aulCritSum = (ULONG *)malloc(XX_NUM_NODES * sizeof(ULONG));
assert(aulCritSum != NULL);
aulN = (ULONG *)malloc(XX_NUM_NODES * sizeof(ULONG));
assert(aulN != NULL);
aulPrio = (ULONG *)malloc(pg->ulN * pg->ulN * sizeof(ULONG));
assert(aulN != NULL);

/* calculate criterion for each body */
for (ulI = 0; ulI < pg->ulN; ulI++)
{
  aulCrit[ulI] = 0;
  for (ulJ = 0; ulJ < pg->ulN; ulJ++)
  {
    aulCrit[ulI] += *(pg->aulBufPrio+ulI*pg->ulN+ulJ);
  }
}

/* initialize counts of every p */
for (ulP = 0; ulP < XX_NUM_NODES; ulP++)
{
  aulCritSum[ulP] = 0;
  aulN[ulP] = 0;
}

/* redistribute bodies on every processor, so that every p
   has same load */
for (ulI = 0; ulI < pg->ulN; ulI++)
{
  ulCritMax = 0;
  /* look for body with greatest criterion */
  for (ulJ = 0; ulJ < pg->ulN; ulJ++)
  {
    if (aulCrit[ulJ] > ulCritMax)
    {
      ulCritMax = aulCrit[ulJ];
      ulIMax = ulJ;
    }
  }
  ulCritSumMin = 2147483647;
  /* look for p with least work so far */
  for (ulP = 0; ulP < XX_NUM_NODES; ulP++)
  {
    if (aulN[ulP] < pg->ulNP &&  aulCritSum[ulP] < ulCritSumMin)
    {
      ulCritSumMin = aulCritSum[ulP];
      ulPMin = ulP;
    }
  }
  /* exchange bodies */
```

```
  ulK = ulPMin * pg->ulNP + aulN[ulPMin];
  bodyTemp = pg->abody[ulK];
  pg->abody[ulK] = pg->abody[ulIMax];
  pg->abody[ulIMax] = bodyTemp;
  for (ulJ = 0; ulJ < pg->ulM; ulJ++)
  {
    fpTemp = pg->afpXSol[ulK*pg->ulM+ulJ];
    pg->afpXSol[ulK*pg->ulM+ulJ] = pg->afpXSol[ulIMax*pg->ulM+ulJ];
    pg->afpXSol[ulIMax*pg->ulM+ulJ] = fpTemp;
  }
  aulCrit[ulIMax] = aulCrit[ulK];
  aulCrit[ulK] = 0;
  aulCritSum[ulPMin] += ulCritMax;
  aulN[ulPMin]++;
}
/* copy old CS matrix to temporary buffer */
for (ulI = 0; ulI < pg->ulN; ulI++)
{
  for (ulJ = 0; ulJ < pg->ulN; ulJ++)
  {
    *(aulPrio+ulI*pg->ulN+ulJ) = *(pg->aulBufPrio+ulI*pg->ulN+ulJ);
  }
}
/* map temporary buffer to new distribution of bodies */
for (ulI = 0; ulI < pg->ulN; ulI++)
{
  for (ulJ = 0; ulJ < pg->ulN; ulJ++)
  {
    *(pg->aulBufPrio+ulI*pg->ulN+ulJ) =
      *(aulPrio + pg->abody[ulI].ulNumber * pg->ulN
        + pg->abody[ulJ].ulNumber);
  }
}
free(aulCrit);
free(aulCritSum);
free(aulN);
free(aulPrio);

/* test output */
ulMin = 2147483647;
ulMax = 0;
ulSum = 0;
for (ulI = 0; ulI < pg->ulN; ulI++)
{
  ulTemp = 0;
  for (ulJ = 0; ulJ < pg->ulN; ulJ++)
  {
    ulTemp += *(pg->aulBufPrio+ulI*pg->ulN+ulJ);
  }
  printf("%3u %3u %5u\n", ulI, pg->abody[ulI].ulNumber, ulTemp);
  ulSum += ulTemp;
```

```
    if ((ulI + 1) % pg->ulNP == 0)
    {
      if (ulSum > ulMax)
      {
        ulMax = ulSum;
      }
      if (ulSum < ulMin)
      {
        ulMin = ulSum;
      }
      printf("%5u\n", ulSum);
      ulSum = 0;
    }
  }
  printf("min = %5u     max = %5u\n", ulMin, ulMax);
}

/*-------------------------------------
   OUTPUTINFO
  -------------------------------------*/
static void OutputInfo(pFileOut)
FILE *pFileOut;
{
  fprintf(pFileOut, "\n#----------------------------------------\n");
  fprintf(pFileOut, "# # particles = %lu\n", pg->ulN);
  fprintf(pFileOut, "# # buffered particles = %lu\n", pg->ulNBuf);
  fprintf(pFileOut, "# # elements/particle = %lu\n", pg->ulM);
  fprintf(pFileOut, "# # iteration steps = %lu\n", pg->ulIterN);
  fprintf(pFileOut, "# # processors = %lu\n",
          XX_NUM_NODES);
  fprintf(pFileOut, "# floating-point precision = %u\n", sizeof(FPTYPE));
  if (fLoadBalance)
  {
    fprintf(pFileOut, "# load balance = yes\n");
  }
  else
  {
    fprintf(pFileOut, "# load balance = no\n");
  }

#ifdef CICO
#  ifdef CICO_X
    fprintf(pFileOut, "# cico=X ");
#  else
    fprintf(pFileOut, "# cico=S ");
#  endif
#else
  fprintf(pFileOut, "# cico=0 ");
#endif
#ifdef LOCK_
  fprintf(pFileOut, "lock=1");
```

```
#else
  fprintf(pFileOut, "lock=0");
#endif
  fprintf(pFileOut, "\n");
#ifdef CONTAINER
  fprintf(pFileOut, "# Container R = %21.12g\n", pg->fpR);
#else
  fprintf(pFileOut, "# No Container\n");
#endif
  fprintf(pFileOut, "# E0 = (%21.12g,%21.12g,%21.12g)\n#\n",
          pg->afpE0[0], pg->afpE0[1], pg->afpE0[2]);
  fprintf(pFileOut, "# In Files = %s\n", szFileIn);
  fprintf(pFileOut, "# In Quad  = %s\n", szFileQuad);
  fprintf(pFileOut, "# In Num   = %s\n", szFileNum);
  fprintf(pFileOut, "# In Geo   = %s\n", szFileGeo);
  fprintf(pFileOut, "# In Sol   = %s\n", szFileInSol);
  fprintf(pFileOut, "# Out Psi  = %s\n", szFileOutPsi);
  fprintf(pFileOut, "# Out Sol  = %s\n", szFileOutSol);
  fprintf(pFileOut, "# Out Err  = %s\n", szFileOutErr);
  fprintf(pFileOut, "# Out Time = %s\n", szFileOutTime);
}

/*-------------------------------------
   OUTPUTPSI
  -------------------------------------*/
static void OutputPsi()
{
  FILE *pFileOutPsi;
  ULONG ulIter, ulK;
  FPTYPE *pfpPsi;

  pFileOutPsi = fopen(szFileOutPsi, "w");
  /* init pointer to psi array */
  pfpPsi = pg->afpPsi;
  /* iteration */
  for (ulIter = 0; ulIter < pg->ulIterN; ulIter++)
  {
    fprintf(pFileOutPsi, "--- Iteration %3hu ---\n", ulIter + 1);
    fprintf(pFileOutPsi,
        "Sphere     Q       X       Y       Z       R              Psi\n");
    for (ulK = 0; ulK < pg->ulN; ulK++)
    {
      fprintf(pFileOutPsi,
              "%4lu %7.3g %7.3g %7.3g %7.3g %7.3g %21.12g\n",
              pg->abody[ulK].ulNumber, pg->abody[ulK].fpQ,
              pg->abody[ulK].afpZ[0],
              pg->abody[ulK].afpZ[1],
              pg->abody[ulK].afpZ[2], pg->abody[ulK].fpR, *pfpPsi);
      pfpPsi++;
    }
    fprintf(pFileOutPsi, "\n");
```

```
  }
  OutputInfo(pFileOutPsi);
  /* close output file */
  fclose(pFileOutPsi);
}

/*------------------------------------
   OUTPUTSOL
  ------------------------------------*/
static void OutputSol()
{
  FILE *pFileOutSol;
  ULONG ulK, ulL;
  ULONG ulI;

  pFileOutSol = fopen(szFileOutSol, "w");
  /* output x on every element */
  for (ulK = 0; ulK < pg->ulN; ulK++)
  {
    /* search for body with original number ulK */
    for (ulL = 0; ulL < pg->ulN; ulL++)
    {
      if (pg->abody[ulL].ulNumber == ulK)
      {
        break;
      }
    }
    for (ulI = ulL * pg->ulM; ulI < (ulL + 1) * pg->ulM; ulI++)
    {
      fprintf(pFileOutSol, "%28.22g\n", pg->afpXOld[ulI]);
    }
  }
  OutputInfo(pFileOutSol);
  /* close output file */
  fclose(pFileOutSol);
}

/*------------------------------------
   OUTPUTERR
  ------------------------------------*/
static void OutputErr()
{
  FILE *pFileOutErr;
  ULONG ulIter;
  FPTYPE fpLambda;
  FPTYPE fpXErr;
#ifdef TIMERS
  VTTYPE vtMin, vtMax;
#endif

  pFileOutErr = fopen(szFileOutErr, "w");
```

```
  fprintf(pFileOutErr,
          "# Iter  Error                    Lambda                Time\n");
#ifdef TIMERS
  VTMinMax(VT_SOLVE, &vtMin, &vtMax);
#endif
  for (ulIter = 0; ulIter < pg->ulIterN; ulIter++)
  {
    fpXErr = pg->afpXErr[ulIter] / (FPTYPE)pg->ulNM;
    fpLambda = pow(fpXErr, 1. / (FPTYPE)(ulIter + 2));
#ifdef TIMERS
    fprintf(pFileOutErr, "%3u %21.12g %21.12g %21.12g\n", ulIter + 1,
            fpXErr, fpLambda,
            (double)vtMax * (double)(ulIter + 1) / (double)pg->ulIterN);
#else
    fprintf(pFileOutErr, "%3u %21.12g %21.12g\n", ulIter + 1,
            fpXErr, fpLambda);
#endif
  }
  OutputInfo(pFileOutErr);
  /* close output file */
  fclose(pFileOutErr);
}

#ifdef TIMERS
/*-------------------------------------
   OUTPUTVT
  -------------------------------------*/
static void OutputVT()
{
  FILE *pFileOutTime;

  /* open output file for append */
  pFileOutTime = fopen(szFileOutTime, "a");
  fprintf(pFileOutTime, "#\n# --- Timers ---\n");
  fprintf(pFileOutTime, "# Timer              min              max\n");
  OutputVT1(pFileOutTime, "Work", VT_WORK);
  OutputVT1(pFileOutTime, "CreateMatrices", VT_CREATEMATRICES);
  OutputVT1(pFileOutTime, "Solve", VT_SOLVE);
  OutputVT1(pFileOutTime, "Communicate Read", VT_SOL_READ);
  OutputVT1(pFileOutTime, "Communicate Write", VT_SOL_WRITE);
  OutputVT1(pFileOutTime, "RecreateMatrices", VT_RECREATEMATRICES);
  OutputVT1(pFileOutTime, "Dotproduct", VT_DOTPRODUCT);
  OutputVT1(pFileOutTime, "CalcPsi", VT_CALCPSI);
  OutputVT1(pFileOutTime, "CalcErr", VT_CALCERR);
  /* close output file */
  fclose(pFileOutTime);
}

/*-------------------------------------
   OUTPUTVT1
  -------------------------------------*/
```

```
static void OutputVT1(pFileOutTime, sz, ulVTSym)
FILE *pFileOutTime;
char *sz;
ULONG ulVTSym;
{
  VTTYPE vtMin, vtMax;

  /* get minimal and maximal time */
  VTMinMax(ulVTSym, &vtMin, &vtMax);
  /* print out */
  fprintf(pFileOutTime, "# %-20s %21.12g %21.12g\n", sz, (double)vtMin,
          (double)vtMax);
}

/*-------------------------------------
   VTMAX
  -------------------------------------*/
static void VTMinMax(ulVTSym, pvtMin, pvtMax)
ULONG ulVTSym;
VTTYPE *pvtMin, *pvtMax;
{
  VTTYPE vtMin, vtMax;
  ULONG ulI;
  VTTYPE vt;

  /* get minimal and maximal time */
  vtMin = pg->avt[0][ulVTSym];
  vtMax = (VTTYPE)0;
  for (ulI = 0; ulI < XX_NUM_NODES; ulI++)
  {
    vt = pg->avt[ulI][ulVTSym];
    if (vt < vtMin)
    {
      vtMin = vt;
    }
    if (vt > vtMax)
    {
      vtMax = vt;
    }
  }
  *pvtMin = vtMin;
  *pvtMax = vtMax;
}
#endif /* TIMERS */

/*----------------------------
   WORK parallel
  ----------------------------*/
void Work()
{
  InitializeWork();
```

```
  VTSTART();
  VTSTART();
  if (ulProc == 0)
  {
    fprintf(pFileStd, "--- CreateMatrices ---\n");
  }
  CreateMatrices();
  VTSTOP(VT_CREATEMATRICES);
  if (ulProc == 0)
  {
    fprintf(pFileStd, "--- Solve ---\n");
  }
  Solve();
  VTSTOP(VT_WORK);
  ExitWork();
}

/*----------------------------------
   INITIALIZEWORK parallel
  ----------------------------------*/
static void InitializeWork()
{
  ULONG ulI;

  /* get node number */
#ifdef WWT
  LOCK(pg->xulProc)
  ulProc = pg->ulProc++;
  UNLOCK(pg->xulProc)
#else
  ulProc = 0;
#endif
  /* row range */
  ulParticleStart = ulProc * pg->ulNP;
  ulParticleStop = (ulProc + 1) * pg->ulNP;
  ulRowStart = ulParticleStart * pg->ulM;
  ulRowStop = ulParticleStop * pg->ulM;
  /* allocate memory for local memory blocks */
  aulABufRef = (ULONG *)malloc(pg->ulNP * pg->ulN * sizeof(ULONG));
  afpA = (FPTYPE *)malloc(pg->ulNMP * pg->ulNBuf * pg->ulM * sizeof(FPTYPE));
  afpB = (FPTYPE *)malloc(pg->ulNMP * sizeof(FPTYPE));
  /* initialize timers */
#ifdef TIMERS
  ulVT = 0;
  for (ulI = 0; ulI < VT_; ulI++)
  {
    avt[ulI] = (VTTYPE)0;
  }
#endif
}
```

```
/*--------------------------------
   EXITWORK parallel
  --------------------------------*/
static void ExitWork()
{
  ULONG ulI;

  /* if output of timers */
#ifdef TIMERS
  if (pg->fOutMode & 8)
  {
    /* copy them to shared memory */
    for (ulI = 0; ulI < VT_; ulI++)
    {
      pg->avt[ulProc][ulI] = avt[ulI];
    }
  }
#endif
}

/*----------------------------
   CREATEMATRICES parallel
  ----------------------------*/
static void CreateMatrices()
{
  ULONG ulK, ulL, ulKK;
  ULONG ulI;
  ULONG ulBufPrioMin;
  ULONG ulLMin;
  ULONG *pulBufPrioK, *pulBufPrioL;
  ULONG *pulABufRefKK, *pulABufRefL;

  /* loop through rows of particles */
  for (ulKK = 0, ulK = ulParticleStart,
       pulBufPrioK = pg->aulBufPrio + ulParticleStart * pg->ulN,
       pulABufRefKK = aulABufRef;
       ulKK < pg->ulNP;
       ulKK++, ulK++, pulBufPrioK += pg->ulN, pulABufRefKK += pg->ulN)
  {
    /* initialize aulABufRef */
    for (ulL = 0, pulABufRefL = pulABufRefKK;
         ulL < pg->ulN;
         ulL++, pulABufRefL++)
    {
      *pulABufRefL = pg->ulN;
    }
    /* loop through buffered submatrices */
    for (ulI = 0; ulI < pg->ulNBuf; ulI++)
    {
      /* store diagonal matrix in any case */
      if (ulI == 0)
```

```
      {
        ulLMin = ulI;
      }
      else
      {
        /* initialize ulBufPrioMin with highest possible priority */
        ulBufPrioMin = pg->ulN * pg->ulN;
        /* search for submatrices with highest priority and buffer these */
        for (ulL = 0, pulBufPrioL = pulBufPrioK, pulABufRefL = pulABufRefKK;
             ulL < pg->ulN;
             ulL++, pulBufPrioL++, pulABufRefL++)
        {
          /* if submatrix is not buffered yet, and its priority is greater
             than the priorities of all other submatrices */
          if (*pulABufRefL == pg->ulN &&
              *pulBufPrioL < ulBufPrioMin)
          {
            ulBufPrioMin = *pulBufPrioL;
            ulLMin = ulL;
          }
        }
      }
      *(pulABufRefKK + ulLMin) = ulI;
      /* check if diagonal block matrix */
      if (ulK == ulLMin)
      {
        /* diagonal block matrix */
        CreateAnn(ulK, afpA + (ulKK * pg->ulNBuf + ulI) * pg->ulM * pg->ulM,
                  pg->ulM);
      }
      else
      {
        /* off-diagonal block matrix */
        CreateAkl(ulK, ulLMin,
                  afpA + (ulKK * pg->ulNBuf + ulI) * pg->ulM * pg->ulM,
                  pg->ulM);
      }
    }
    /* create RHS subvector */
    CreateBn(ulK, &afpB[ulKK * pg->ulM]);
  }
}

#ifdef CONTAINER
/*-------------------------------------
   CREATEANN parallel
   ulN - particle number
   pfpA - position of submatrix
   ulIncr - increment from row to row
  -------------------------------------*/
static void CreateAnn(ulN, pfpA, ulIncr)
```

```
ULONG ulN;
FPTYPE *pfpA;
ULONG ulIncr;
{
  static ULONG ulI, ulJ;
  static USHORT usIXs, usIXt;
  static FPTYPE afpXmY[3];
  static FPTYPE fpRxy2;
  static FPTYPE fpRxy;
  static FPTYPE fpSum;
  static FPTYPE afpX[3];
  static FPTYPE afpY[3];
  static FPTYPE *pfpAA, *pfpAAA;
  static PGEO pgeoI, pgeoJ;
  static PBODY pbodyN;
  static FPTYPE fpS1;
  static FPTYPE fpNR2;
  static ULONG ulII, ulJJ;
  static FPTYPE fpRy;
  static FPTYPE afpD[3];
  static FPTYPE fpRz;
  static FPTYPE afpZ[3];
  static FPTYPE afpXmZ[3];
  static FPTYPE fpRxz;
  static FPTYPE afpSum1[3];

  pbodyN = &pg->abody[ulN];

  fpNR2 = SQR(pbodyN->fpR);
  fpS1 = .5 * fpNR2 / PI;

  /* loop through rows */
  for (ulI = 0, pfpAA = pfpA; ulI < pg->ulM; ulI++, pfpAA += ulIncr)
  {
    pgeoI = &pg->ageo[ulI];
    afpX[0] = pgeoI->afpX[0] * pbodyN->fpR + pbodyN->afpZ[0];
    afpX[1] = pgeoI->afpX[1] * pbodyN->fpR + pbodyN->afpZ[1];
    afpX[2] = pgeoI->afpX[2] * pbodyN->fpR + pbodyN->afpZ[2];
    /* loop through columns */
    for (ulJ = 0, pfpAAA = pfpAA;
         ulJ < pg->ulM;
         ulJ++, pfpAAA++)
    {
      pgeoJ = &pg->ageo[ulJ];
      afpY[0] = pgeoI->afpX[0] * pbodyN->fpR + pbodyN->afpZ[0];
      afpY[1] = pgeoI->afpX[1] * pbodyN->fpR + pbodyN->afpZ[1];
      afpY[2] = pgeoI->afpX[2] * pbodyN->fpR + pbodyN->afpZ[2];
      afpXmY[0] = afpX[0] - afpY[0];
      afpXmY[1] = afpX[1] - afpY[1];
      afpXmY[2] = afpX[2] - afpY[2];
      fpRxy2 = afpXmY[0]*afpXmY[0] + afpXmY[1]*afpXmY[1] +
```

```
            afpXmY[2]*afpXmY[2];
      /* check if large distance */
      if (fpRxy2 > 18. * pgeoJ->fpDA * fpNR2)
      {
        /* 1-point Gaussian-Legendre Quadrature */
        fpRxy = sqrt(fpRxy2);
        CONTAINER_1
        *pfpAAA = -pgeoJ->fpDA / pg->fpGeoA -
                  fpS1 * pgeoJ->fpDA *
                  CONTAINER_2
      } /* check if large distance */
      else
      {
        /* Gaussian-Legendre Quadrature */
        fpSum = 0.;
        for (usIXs = 0; usIXs < pg->usQuadN; usIXs++)
        {
          for (usIXt = 0; usIXt < pg->usQuadN; usIXt++)
          {
            afpY[0] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[0] +
                      *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                      pgeoJ->afpU2[0] +
                      pgeoJ->afpP2[0]) * pbodyN->fpR + pbodyN->afpZ[0];
            afpY[1] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[1] +
                      *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                      pgeoJ->afpU2[1] +
                      pgeoJ->afpP2[1]) * pbodyN->fpR + pbodyN->afpZ[1];
            afpY[2] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[2] +
                      *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                      pgeoJ->afpU2[2] +
                      pgeoJ->afpP2[2]) * pbodyN->fpR + pbodyN->afpZ[2];
            afpXmY[0] = afpX[0] - afpY[0];
            afpXmY[1] = afpX[1] - afpY[1];
            afpXmY[2] = afpX[2] - afpY[2];
            fpRxy = AABS(afpXmY);
            CONTAINER_1
            fpSum += pg->afpQuadW[usIXs] *
                     *(pg->afpQuadWt+usIXs*pg->usQuadN+usIXt) *
                     pgeoJ->fpJacob *
                     CONTAINER_2
          }
        }
        *pfpAAA = -pgeoJ->fpDA / pg->fpGeoA - fpSum * fpS1;
      } /* check if large distance */
    } /* loop through columns */
  } /* loop through rows */
}

/*-------------------------------------
   CREATEAKL parallel
   ulK - particle number (x)
```

```
   ulL - particle number (y)
   pfpA - position of submatrix
   ulIncr - increment from row to row
  -------------------------------------*/
static void CreateAkl(ulK, ulL, pfpA, ulIncr)
ULONG ulK;
ULONG ulL;
FPTYPE *pfpA;
ULONG ulIncr;
{
  static ULONG ulI, ulJ;
  static USHORT usIXs, usIXt;
  static FPTYPE afpXmY[3];
  static FPTYPE fpRxy2;
  static FPTYPE fpRxy;
  static FPTYPE fpSum;
  static FPTYPE afpX[3];
  static FPTYPE afpY[3];
  static FPTYPE *pfpAA, *pfpAAA;
  static PGEO pgeoI, pgeoJ;
  static PBODY pbodyK, pbodyL;
  static FPTYPE fpS1;
  static FPTYPE fpLR2;
  static ULONG ulII, ulJJ;
  static FPTYPE fpRy;
  static FPTYPE afpD[3];
  static FPTYPE fpRz;
  static FPTYPE afpZ[3];
  static FPTYPE afpXmZ[3];
  static FPTYPE fpRxz;
  static FPTYPE afpSum1[3];

  pbodyK = &pg->abody[ulK];
  pbodyL = &pg->abody[ulL];

  fpLR2 = SQR(pbodyL->fpR);
  fpS1 = .5 * fpLR2 / PI;

  /* loop through rows */
  for (ulI = 0, pfpAA = pfpA; ulI < pg->ulM; ulI++, pfpAA += ulIncr)
  {
    pgeoI = &pg->ageo[ulI];
    afpX[0] = pgeoI->afpX[0] * pbodyK->fpR + pbodyK->afpZ[0];
    afpX[1] = pgeoI->afpX[1] * pbodyK->fpR + pbodyK->afpZ[1];
    afpX[2] = pgeoI->afpX[2] * pbodyK->fpR + pbodyK->afpZ[2];
    /* loop through columns */
    for (ulJ = 0, pfpAAA = pfpAA;
         ulJ < pg->ulM;
         ulJ++, pfpAAA++)
    {
      pgeoJ = &pg->ageo[ulJ];
```

```
      afpY[0] = pgeoI->afpX[0] * pbodyL->fpR + pbodyL->afpZ[0];
      afpY[1] = pgeoI->afpX[1] * pbodyL->fpR + pbodyL->afpZ[1];
      afpY[2] = pgeoI->afpX[2] * pbodyL->fpR + pbodyL->afpZ[2];
      afpXmY[0] = afpX[0] - afpY[0];
      afpXmY[1] = afpX[1] - afpY[1];
      afpXmY[2] = afpX[2] - afpY[2];
      fpRxy2 = afpXmY[0]*afpXmY[0] + afpXmY[1]*afpXmY[1] +
               afpXmY[2]*afpXmY[2];
      /* check if large distance */
      if (fpRxy2 > 18. * pgeoJ->fpDA * fpLR2)
      {
        /* 1-point Gaussian-Legendre Quadrature */
        fpRxy = sqrt(fpRxy2);
        CONTAINER_1
        *pfpAAA = -fpS1 * pgeoJ->fpDA *
                  CONTAINER_2
      } /* check if large distance */
      else
      {
        /* Gaussian-Legendre Quadrature */
        fpSum = 0.;
        for (usIXs = 0; usIXs < pg->usQuadN; usIXs++)
        {
/* FRANK: factor out w_s */
          for (usIXt = 0; usIXt < pg->usQuadN; usIXt++)
          {
            afpY[0] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[0] +
                       *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                       pgeoJ->afpU2[0] +
                       pgeoJ->afpP2[0]) * pbodyL->fpR + pbodyL->afpZ[0];
            afpY[1] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[1] +
                       *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                       pgeoJ->afpU2[1] +
                       pgeoJ->afpP2[1]) * pbodyL->fpR + pbodyL->afpZ[1];
            afpY[2] = (pg->afpQuadX[usIXs] * pgeoJ->afpU1[2] +
                       *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                       pgeoJ->afpU2[2] +
                       pgeoJ->afpP2[2]) * pbodyL->fpR + pbodyL->afpZ[2];
            afpXmY[0] = afpX[0] - afpY[0];
            afpXmY[1] = afpX[1] - afpY[1];
            afpXmY[2] = afpX[2] - afpY[2];
            fpRxy = AABS(afpXmY);
            CONTAINER_1
            fpSum += pg->afpQuadW[usIXs] *
                     *(pg->afpQuadWt+usIXs*pg->usQuadN+usIXt) *
                     pgeoJ->fpJacob *
                     CONTAINER_2
          }
        }
        *pfpAAA = -fpSum * fpS1;
      } /* check if large distance */
```

```
    } /* loop through columns */
  } /* loop through rows */
}

#else

/*-------------------------------------
   CREATEANN parallel
   ulN - particle number
   pfpA - position of submatrix
   ulIncr - increment from row to row
  -------------------------------------*/
static void CreateAnn(ulN, pfpA, ulIncr)
ULONG ulN;
FPTYPE *pfpA;
ULONG ulIncr;
{
  static ULONG ulI, ulJ;
  static USHORT usIXs, usIXt;
  static FPTYPE afpXmY[3];
  static FPTYPE fpR;
  static FPTYPE fpRxy2;
  static FPTYPE fpRxy;
  static FPTYPE fpSum;
  static FPTYPE afpY[3];
  static FPTYPE *pfpAA, *pfpAAA;
  static PGEO pgeoI, pgeoJ;

  /* radius of particle ulN */
  fpR = pg->abody[ulN].fpR;

  /* loop through rows */
  for (ulI = 0, pfpAA = pfpA; ulI < pg->ulM; ulI++, pfpAA += ulIncr)
  {
    pgeoI = &pg->ageo[ulI];
    /* loop through columns */
    for (ulJ = 0, pfpAAA = pfpAA;
         ulJ < pg->ulM;
         ulJ++, pfpAAA++)
    {
      pgeoJ = &pg->ageo[ulJ];
      /* check if diagonal element */
      if (ulJ == ulI)
      {
        *pfpAAA = -pgeoI->fpDA / pg->fpGeoA;
      } /* check if diagonal element */
      else
      {
        afpXmY[0] = (pgeoI->afpX[0] - pgeoJ->afpX[0]) *
                    fpR;
        afpXmY[1] = (pgeoI->afpX[1] - pgeoJ->afpX[1]) *
```

```
            fpR;
afpXmY[2] = (pgeoI->afpX[2] - pgeoJ->afpX[2]) *
            fpR;
fpRxy2 = afpXmY[0]*afpXmY[0] + afpXmY[1]*afpXmY[1] +
         afpXmY[2]*afpXmY[2];
/* check if large distance */
if (fpRxy2 > 18. * pgeoJ->fpDA * fpR * fpR)
{
  /* no Gaussian-Legendre Quadrature */
  fpRxy = sqrt(fpRxy2);
  *pfpAAA = -pgeoJ->fpDA * fpR * fpR *
          .5 * (pgeoJ->afpN[0]*afpXmY[0] +
          pgeoJ->afpN[1]*afpXmY[1] +
          pgeoJ->afpN[2]*afpXmY[2]) /
          (PI * fpRxy * fpRxy * fpRxy) -
          pgeoJ->fpDA / pg->fpGeoA;
} /* check if large distance */
else
{
  /* Gaussian-Legendre Quadrature */
  fpSum = 0.;
  for (usIXs = 0; usIXs < pg->usQuadN; usIXs++)
  {
    for (usIXt = 0; usIXt < pg->usQuadN; usIXt++)
    {
      afpY[0] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[0] +
                *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                pgeoJ->afpU2[0] +
                pgeoJ->afpP2[0];
      afpY[1] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[1] +
                *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                pgeoJ->afpU2[1] +
                pgeoJ->afpP2[1];
      afpY[2] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[2] +
                *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                pgeoJ->afpU2[2] +
                pgeoJ->afpP2[2];
      afpXmY[0] = (pgeoI->afpX[0] - afpY[0]) * fpR;
      afpXmY[1] = (pgeoI->afpX[1] - afpY[1]) * fpR;
      afpXmY[2] = (pgeoI->afpX[2] - afpY[2]) * fpR;
      fpRxy = sqrt(afpXmY[0]*afpXmY[0] +
                   afpXmY[1]*afpXmY[1] +
                   afpXmY[2]*afpXmY[2]);
      fpSum += pg->afpQuadW[usIXs] *
               *(pg->afpQuadWt+usIXs*pg->usQuadN+usIXt) *
               pgeoJ->fpJacob *
               (pgeoJ->afpN[0]*afpXmY[0] +
               pgeoJ->afpN[1]*afpXmY[1] +
               pgeoJ->afpN[2]*afpXmY[2]) /
               (fpRxy * fpRxy * fpRxy);
    }
```

```
          }
          *pfpAAA = -fpSum * .5 * fpR * fpR / PI -
                   pgeoJ->fpDA / pg->fpGeoA;
        } /* check if large distance */
      } /* check if diagonal element */
    } /* loop through columns */
  } /* loop through rows */
}

/*-------------------------------------
   CREATEAKL parallel
   ulK - particle number (x)
   ulL - particle number (y)
   pfpA - position of submatrix
   ulIncr - increment from row to row
  -------------------------------------*/
static void CreateAkl(ulK, ulL, pfpA, ulIncr)
ULONG ulK;
ULONG ulL;
FPTYPE *pfpA;
ULONG ulIncr;
{
  static ULONG ulI, ulJ;
  static USHORT usIXs, usIXt;
  static FPTYPE afpXmY[3];
  static FPTYPE fpRxy2;
  static FPTYPE fpRxy;
  static FPTYPE fpSum;
  static FPTYPE afpX[3];
  static FPTYPE afpY[3];
  static FPTYPE *pfpAA, *pfpAAA;
  static FPTYPE afpZdst[3];
  static PGEO pgeoI, pgeoJ;
  static PBODY pbodyK, pbodyL;

  pbodyK = &pg->abody[ulK];
  pbodyL = &pg->abody[ulL];
  /* distance between particle centers */
  afpZdst[0] = pbodyK->afpZ[0] - pbodyL->afpZ[0];
  afpZdst[1] = pbodyK->afpZ[1] - pbodyL->afpZ[1];
  afpZdst[2] = pbodyK->afpZ[2] - pbodyL->afpZ[2];

  /* loop through rows */
  for (ulI = 0, pfpAA = pfpA; ulI < pg->ulM; ulI++, pfpAA += ulIncr)
  {
    pgeoI = &pg->ageo[ulI];
    afpX[0] = pgeoI->afpX[0] * pbodyK->fpR;
    afpX[1] = pgeoI->afpX[1] * pbodyK->fpR;
    afpX[2] = pgeoI->afpX[2] * pbodyK->fpR;
    /* loop through columns */
    for (ulJ = 0, pfpAAA = pfpAA;
```

```
    ulJ < pg->ulM;
    ulJ++, pfpAAA++)
{
  pgeoJ = &pg->ageo[ulJ];
  afpXmY[0] = afpX[0] -
              pgeoJ->afpX[0] * pbodyL->fpR +
              afpZdst[0];
  afpXmY[1] = afpX[1] -
              pgeoJ->afpX[1] * pbodyL->fpR +
              afpZdst[1];
  afpXmY[2] = afpX[2] -
              pgeoJ->afpX[2] * pbodyL->fpR +
              afpZdst[2];
  fpRxy2 = afpXmY[0]*afpXmY[0] + afpXmY[1]*afpXmY[1] +
           afpXmY[2]*afpXmY[2];
  /* check if large distance */
  if (fpRxy2 > 18. * pgeoJ->fpDA * pbodyL->fpR * pbodyL->fpR)
  {
    /* no Gaussian-Legendre Quadrature */
    fpRxy = sqrt(fpRxy2);
    *pfpAAA = -pgeoJ->fpDA * pbodyL->fpR * pbodyL->fpR *
            .5 * (pgeoJ->afpN[0]*afpXmY[0] +
            pgeoJ->afpN[1]*afpXmY[1] +
            pgeoJ->afpN[2]*afpXmY[2]) /
            (PI * fpRxy * fpRxy * fpRxy);
  } /* check if large distance */
  else
  {
    /* Gaussian-Legendre Quadrature */
    fpSum = 0.;
    for (usIXs = 0; usIXs < pg->usQuadN; usIXs++)
    {
      for (usIXt = 0; usIXt < pg->usQuadN; usIXt++)
      {
        afpY[0] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[0] +
                  *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                  pgeoJ->afpU2[0] +
                  pgeoJ->afpP2[0];
        afpY[1] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[1] +
                  *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                  pgeoJ->afpU2[1] +
                  pgeoJ->afpP2[1];
        afpY[2] = pg->afpQuadX[usIXs] * pgeoJ->afpU1[2] +
                  *(pg->afpQuadXt+usIXs*pg->usQuadN+usIXt) *
                  pgeoJ->afpU2[2] +
                  pgeoJ->afpP2[2];
        afpXmY[0] = afpX[0] -
                    afpY[0] * pbodyL->fpR +
                    afpZdst[0];
        afpXmY[1] = afpX[1] -
                    afpY[1] * pbodyL->fpR +
```

```
                    afpZdst[1];
            afpXmY[2] = afpX[2] -
                    afpY[2] * pbodyL->fpR +
                    afpZdst[2];
            fpRxy = sqrt(afpXmY[0]*afpXmY[0] +
                     afpXmY[1]*afpXmY[1] +
                     afpXmY[2]*afpXmY[2]);
            fpSum += pg->afpQuadW[usIXs] *
                     *(pg->afpQuadWt+usIXs*pg->usQuadN+usIXt) *
                     pgeoJ->fpJacob *
                     (pgeoJ->afpN[0]*afpXmY[0] +
                     pgeoJ->afpN[1]*afpXmY[1] +
                     pgeoJ->afpN[2]*afpXmY[2]) /
                     (fpRxy * fpRxy * fpRxy);
          }
        }
        *pfpAAA = -fpSum * .5 * pbodyL->fpR * pbodyL->fpR / PI;
      } /* check if large distance */
    } /* loop through columns */
  } /* loop through rows */
}
#endif

/*-------------------------------------
   CREATEBN parallel
   ulN - particle number
   pfpB - position of subvector
  -------------------------------------*/
static void CreateBn(ulK, pfpB)
ULONG ulK;
FPTYPE *pfpB;
{
  static ULONG ulI;
  static ULONG ulL;
  static FPTYPE fpSum;
  static FPTYPE afpX[3];
  static FPTYPE afpXmZ[3];
  static FPTYPE *pfpBB;
  static PBODY pbodyK, pbodyL;
#ifdef CONTAINER
  static FPTYPE fpRz, fpRy;
  static FPTYPE afpD[3];
#endif

  pbodyK = &pg->abody[ulK];
  /* loop through rows */
  for (ulI = 0, pfpBB = pfpB; ulI < pg->ulM; ulI++, pfpBB++)
  {
    afpX[0] = pg->ageo[ulI].afpX[0] * pbodyK->fpR + pbodyK->afpZ[0];
    afpX[1] = pg->ageo[ulI].afpX[1] * pbodyK->fpR + pbodyK->afpZ[1];
    afpX[2] = pg->ageo[ulI].afpX[2] * pbodyK->fpR + pbodyK->afpZ[2];
```

```
    fpSum = 0.;
    /* loop through particles */
    for (ulL = 0; ulL < pg->ulN; ulL++)
    {
      pbodyL = &pg->abody[ulL];
      afpXmZ[0] = afpX[0] - pbodyL->afpZ[0];
      afpXmZ[1] = afpX[1] - pbodyL->afpZ[1];
      afpXmZ[2] = afpX[2] - pbodyL->afpZ[2];
      fpSum += pbodyL->fpQ / AABS(afpXmZ);
#ifdef CONTAINER
      fpRy = AABS(pbodyL->afpZ);
      afpD[0] = pbodyL->afpZ[0] / fpRy;
      afpD[1] = pbodyL->afpZ[1] / fpRy;
      afpD[2] = pbodyL->afpZ[2] / fpRy;
      fpRz = pg->fpR2 / fpRy;
      afpXmZ[0] = afpX[0] - afpD[0] * fpRz;
      afpXmZ[1] = afpX[1] - afpD[1] * fpRz;
      afpXmZ[2] = afpX[2] - afpD[2] * fpRz;
      fpSum -= pbodyL->fpQ * fpRz / (AABS(afpXmZ) * pg->fpR);
#endif
    }
    *pfpBB = -fpSum + pg->afpE0[0] * afpX[0] + pg->afpE0[1] * afpX[1] +
             pg->afpE0[2] * afpX[2];
  }
}

/*-------------------------------------
   SOLVE parallel
  -------------------------------------*/
static void Solve()
{
  ULONG ulI, ulII, ulJJ, ulK, ulKK, ulL;
  ULONG ulIter;
  FPTYPE fpSum;
  FPTYPE *afpX;
  FPTYPE *afpXCom;
  FPTYPE *afpATemp;           /* non buffered temporary submatrix */
  FPTYPE *pfpA;               /* pointer to submatrix */
  FPTYPE *afpXBuf;
  FPTYPE *pfpPsi, *pfpPsi1;
  FPTYPE *pfpXBuf, *pfpXBuf1;
  FPTYPE *afpXErr;
  FPTYPE *pfpXErr;
#ifdef WWT
  ULONG ulBarrier;
#endif
  ULONG *pulABufRefKK, *pulABufRefL;
  ULONG *pulBufPrioK, *pulBufPrioL;
  FPTYPE *pfpXCom, *pfpXOld;
  FPTYPE *pfpXKK, *pfpXII;
  FPTYPE *pfpXSol;
```

```
  /* allocate memory for local solution vector */
  afpX = (FPTYPE *)malloc(pg->ulNMP * sizeof(FPTYPE));
  /* allocate memory for received data */
  afpXCom = (FPTYPE *)malloc(pg->ulM * sizeof(FPTYPE));
  /* allocate memory for buffer solution matrix */
  afpXBuf = (FPTYPE *)malloc(pg->ulNMP * pg->ulN * sizeof(FPTYPE));
  for (ulI = 0, pfpXBuf = afpXBuf; ulI < pg->ulNMP * pg->ulN; ulI++, pfpXBuf++)
  {
    *pfpXBuf = 0.;
  }

  /* allocate memory for temporary sub matrix of a */
  afpATemp = (FPTYPE *)malloc(pg->ulM * pg->ulM * sizeof(FPTYPE));

  /* if output of psi */
  if (pg->fOutMode & 1)
  {
    /* pfpPsi points to first element in pg->afpPsi */
    pfpPsi1 = &pg->afpPsi[ulParticleStart];
  }
  /* if output of solution error */
  if (pg->fOutMode & 4)
  {
    /* allocate memory for locally stored error */
    afpXErr = (FPTYPE *)malloc(pg->ulIterN * sizeof(FPTYPE));
    /* pfpXErr points to first element in afpXErr */
    pfpXErr = afpXErr;
  }

  /* initial guess */
  for (ulII = 0, ulI = ulRowStart; ulII < pg->ulNMP; ulII++, ulI++)
  {
    pg->afpXOld[ulI] = afpX[ulII] = afpB[ulII];
  }

#ifdef WWT
  /* synchronize all nodes */
  BARRIER(ulBarrier, XX_NUM_NODES)
#endif

  VTSTART();
  /* iteration */
  for (ulIter = 0; ulIter < pg->ulIterN; ulIter++)
  {
    /* if on proc 0 */
/*
    if (ulProc == 0)
    {
      fprintf(pFileStd, "Iteration %u\n", ulIter);
    }
```

```
*/
    /* if output of psi */
    if (pg->fOutMode & 1)
    {
      pfpPsi = pfpPsi1;
      pfpPsi1 += pg->ulN;
    }
    /* pfpXBuf points to first element in afpXBuf */
    pfpXBuf = afpXBuf;
    /* loop through rows (particles) */
    for (ulKK = 0, ulK = ulParticleStart, pulABufRefKK = aulABufRef,
         pulBufPrioK = pg->aulBufPrio + ulParticleStart * pg->ulN,
         pfpXKK = afpX;
         ulKK < pg->ulNP;
         ulKK++, ulK++, pulABufRefKK += pg->ulN, pulBufPrioK += pg->ulN,
         pfpXKK += pg->ulM)
    {
      /* loop through columns (particles) */
      for (ulL = 0, pulABufRefL = pulABufRefKK, pulBufPrioL = pulBufPrioK;
           ulL < pg->ulN;
           ulL++, pulABufRefL++, pulBufPrioL++)
      {
        /* if sub solution vector is to be updated */
        if (ulIter % *pulBufPrioL == 0)
        {
          /* if submatrix is not buffered */
          if (*pulABufRefL == pg->ulN)
          {
            VTSTART();
            /* create submatrix */
            /* check if diagonal block matrix */
            if (ulK == ulL)
            {
              /* diagonal block matrix */
              CreateAnn(ulK, afpATemp, pg->ulM);
            }
            else
            {
              /* off-diagonal block matrix */
              CreateAkl(ulK, ulL, afpATemp, pg->ulM);
            }
            VTSTOP(VT_RECREATEMATRICES);
            pfpA = afpATemp;
          } /* if submatrix is not buffered */
          else
          {
            pfpA = afpA + (ulKK * pg->ulNBuf + *pulABufRefL) *
                   pg->ulM * pg->ulM;
          }
          /* get global old sub solution vector */
          VTSTART();
```

```
#ifdef LOCK_
          LOCK(pg->axafpXOld[ulL])
#endif
          for (ulJJ = 0, pfpXCom = afpXCom,
               pfpXOld = pg->afpXOld + ulL * pg->ulM;
               ulJJ < pg->ulM;
               ulJJ++, pfpXCom++, pfpXOld++)
          {
#ifdef CICO
#  ifdef CICO_X
            ALIGNED_CO_X(pfpXOld);
#  else
            ALIGNED_CO_S(pfpXOld);
#  endif
#endif
            *pfpXCom = *pfpXOld;
#ifdef CICO
            ALIGNED_CI(pfpXOld, sizeof(FPTYPE));
#endif
          }
/*
          for (ulJ = 0, pfpXCom = afpXCom,
               pfpXOld = pg->afpXOld + ulL * pg->ulM;
               ulJ < pg->ulM;
               ulJ += ALIGN_FACTOR)
          {
            CO_X(pfpXOld)
            for (ulJJ = ulJ;
                 ulJJ < ulJ + ALIGN_FACTOR && ulJJ < pg->ulM;
                 ulJJ++, pfpXCom++, pfpXOld++)
            {
              *pfpXCom = *pfpXOld;
            }
            CI(pfpXOld - 1)
          }
*/
#ifdef LOCK_
          UNLOCK(pg->axafpXOld[ulL])
#endif
          VTSTOP(VT_SOL_READ);
          VTSTART();
          /* loop thru rows */
          for (ulII = 0, pfpXBuf1 = pfpXBuf, pfpXII = pfpXKK;
               ulII < pg->ulM;
               ulII++, pfpXBuf1++, pfpXII++)
          {
            fpSum = 0.;
            /* loop thru cols */
            for (ulJJ = 0, pfpXCom = afpXCom;
                 ulJJ < pg->ulM;
                 ulJJ++, pfpXCom++)
```

```
            {
              fpSum += *(pfpA++) * *pfpXCom;
            }
            *pfpXII += fpSum - *pfpXBuf1;
            *pfpXBuf1 = fpSum;
          }
          VTSTOP(VT_DOTPRODUCT);
        } /* if sub solution vector is to be updated */
        pfpXBuf += pg->ulM;
      } /* loop through columns (particles) */
      /* copy solution to global old vector */
      VTSTART();
#ifdef LOCK_
      LOCK(pg->axafpXOld[ulK])
#endif
      for (ulII = 0, pfpXII = pfpXKK,
           pfpXOld = pg->afpXOld + ulK * pg->ulM;
           ulII < pg->ulM;
           ulII++, pfpXII++, pfpXOld++)
      {
#ifdef CICO
        ALIGNED_CO_X(pfpXOld);
#endif
        *pfpXOld = *pfpXII;
#ifdef CICO
        ALIGNED_CI(pfpXOld, sizeof(FPTYPE));
#endif
      }
/*
      for (ulI = 0, pfpXII = pfpXKK,
           pfpXOld = pg->afpXOld + ulK * pg->ulM;
           ulI < pg->ulM;
           ulI += ALIGN_FACTOR)
      {
        CO_X(pfpXOld)
        for (ulII = ulI;
             ulII < ulI + ALIGN_FACTOR && ulII < pg->ulM;
             ulII++, pfpXOld++, pfpXII++)
        {
          *pfpXOld = *pfpXII;
        }
        CI(pfpXOld - 1)
      }
*/
#ifdef LOCK_
      UNLOCK(pg->axafpXOld[ulK])
#endif
      VTSTOP(VT_SOL_WRITE);
      /* if output of psi */
      if (pg->fOutMode & 1)
      {
```

```
        /* compute surface potentials */
        VTSTART();
        fpSum = 0.;
        for (ulII = 0, pfpXII = pfpXKK; ulII < pg->ulM; ulII++, pfpXII++)
        {
          fpSum += *pfpXII * pg->ageo[ulII].fpDA;
        }
        *pfpPsi = -fpSum / pg->fpGeoA;
        pfpPsi++;
        VTSTOP(VT_CALCPSI);
      }
    } /* loop through rows (particles) */
    /* if output of solution error */
    if (pg->fOutMode & 4)
    {
      /* compute error on current processor */
      VTSTART();
      fpSum = 0.;
      for (ulII = 0, pfpXII = afpX, pfpXSol = pg->afpXSol + ulRowStart;
           ulII < pg->ulNMP;
           ulII++, pfpXII++, pfpXSol++)
      {
        fpSum += fabs((*pfpXII - *pfpXSol) / *pfpXSol);
      }
      *(pfpXErr++) = fpSum;
      VTSTOP(VT_CALCERR);
    }
  } /* iteration */
  VTSTOP(VT_SOLVE);

  /* if output of solution error */
  if (pg->fOutMode & 4)
  {
    /* copy it to shared memory */
#ifdef WWT
    /* lock vector */
    LOCK(pg->xafpXErr)
#endif
    /* add local error to global error */
    pfpXErr = afpXErr;
    for (ulIter = 0; ulIter < pg->ulIterN; ulIter++)
    {
      pg->afpXErr[ulIter] += *(pfpXErr++);
    }
#ifdef WWT
    /* unlock it */
    UNLOCK(pg->xafpXErr)
#endif
    /* free local memory */
    free(afpXErr);
  }
```

```
  /* free memory */
  free(aulABufRef);
  free(afpA);
  free(afpB);
  free(afpXBuf);
  free(afpX);
  free(afpXCom);
  free(afpATemp);
}
```

B.5.2 Module geometry.c

```
/*---------------------------------*/
/* geometry.c                      */
/*---------------------------------*/
/* module to be used by            */
/* lap.c                           */
/*---------------------------------*/
/* programmed by Frank Traenkle    */
/*---------------------------------*/
/* creates boundary elements for   */
/* sphere of radius 1              */
/*---------------------------------*/

/*-----------------------------*/
/* INCLUDEs                    */
/*-----------------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "../tools/my.h"
#include "geometry.h"

/*-----------------------------*/
/* DEFINEs                     */
/*-----------------------------*/
#define M_MAX 320

/*-----------------------------*/
/* PROTOTYPEs                  */
/*-----------------------------*/
static void CreateTetrahedron();
static void CreateOctahedron();
static void CreateIcosahedron();
static void Tesselate();
static void Bisector();
```

```
static void NewElements();
static void AddElement();
static void RenameElement();

/*----------------------------*/
/* VARIABLEs                  */
/*----------------------------*/
static FPTYPE (*afpNode)[3];
static USHORT (*ausElement)[3];
static USHORT usLastElement;
static USHORT usLastNode;
static USHORT ausNewNode[3];

/*----------------------------*/
/* FUNCTION CreateGeometry    */
/*----------------------------*/
extern USHORT CreateGeometry(usMstart, usNtess)
USHORT usMstart;
USHORT usNtess;
{
  USHORT usI;
  USHORT usM;

  /* check if number of elements is too high */
  usM = usMstart;
  for(usI = 0; usI < usNtess; usI++)
  {
    usM *= 4;
  }
  if (usM > M_MAX)
  {
    /* not enough memory for this order of tesselation */
    return 0;
  }

  /* allocate memory */
  afpNode = (FPTYPE (*)[3])malloc(usM * sizeof(*afpNode));
  ausElement = (USHORT (*)[3])malloc(usM * sizeof(*ausElement));

  /* create starting polyhedron */
  switch(usMstart)
  {
    case 4:
      CreateTetrahedron();
      break;
    case 8:
      CreateOctahedron();
      break;
    case 20:
      CreateIcosahedron();
      break;
```

```
    default:
      /* usMstart is not valid */
      return 0;
  }

  /* tesselate usNtess times */
  for(usI = 0; usI < usNtess; usI++)
  {
    Tesselate();
  }

  return usM;
}

/*----------------------------*/
/* FUNCTION CloseGeometry     */
/*----------------------------*/
extern void CloseGeometry()
{
  free(afpNode);
  free(ausElement);
}

/*----------------------------*/
/* FUNCTION CreateTetrahedron */
/*----------------------------*/
static void CreateTetrahedron()
{
  FPTYPE fpTheta;

  fpTheta = PI - acos(1. / 3.);

  afpNode[0][0] = 0.0;
  afpNode[0][1] = 0.0;
  afpNode[0][2] = 1.0;

  afpNode[1][0] = sin(fpTheta) * sin(0.0);
  afpNode[1][1] = sin(fpTheta) * cos(0.0);
  afpNode[1][2] = cos(fpTheta);

  afpNode[2][0] = sin(fpTheta) * sin(2. / 3. * PI);
  afpNode[2][1] = sin(fpTheta) * cos(2. / 3. * PI);
  afpNode[2][2] = cos(fpTheta);

  afpNode[3][0] = sin(fpTheta) * sin(4. / 3. * PI);
  afpNode[3][1] = sin(fpTheta) * cos(4. / 3. * PI);
  afpNode[3][2] = cos(fpTheta);

  ausElement[0][0] = 0;
  ausElement[0][1] = 1;
  ausElement[0][2] = 2;
```

```
  ausElement[1][0] = 0;
  ausElement[1][1] = 1;
  ausElement[1][2] = 3;
  ausElement[2][0] = 0;
  ausElement[2][1] = 2;
  ausElement[2][2] = 3;
  ausElement[3][0] = 1;
  ausElement[3][1] = 2;
  ausElement[3][2] = 3;

  usLastElement = 4;
  usLastNode = 4;
}

/*----------------------------*/
/* FUNCTION CreateOctahedron  */
/*----------------------------*/
static void CreateOctahedron()
{
  afpNode[0][0] = 0.0;
  afpNode[0][1] = 0.0;
  afpNode[0][2] = 1.0;

  afpNode[1][0] = 1.0;
  afpNode[1][1] = 0.0;
  afpNode[1][2] = 0.0;

  afpNode[2][0] = 0.0;
  afpNode[2][1] = 1.0;
  afpNode[2][2] = 0.0;

  afpNode[3][0] = -1.0;
  afpNode[3][1] = 0.0;
  afpNode[3][2] = 0.0;

  afpNode[4][0] = 0.0;
  afpNode[4][1] = -1.0;
  afpNode[4][2] = 0.0;

  afpNode[5][0] = 0.0;
  afpNode[5][1] = 0.0;
  afpNode[5][2] = -1.0;

  ausElement[0][0] = 0;
  ausElement[0][1] = 1;
  ausElement[0][2] = 2;
  ausElement[1][0] = 0;
  ausElement[1][1] = 2;
  ausElement[1][2] = 3;
  ausElement[2][0] = 0;
  ausElement[2][1] = 3;
```

```
  ausElement[2][2] = 4;
  ausElement[3][0] = 0;
  ausElement[3][1] = 4;
  ausElement[3][2] = 1;
  ausElement[4][0] = 5;
  ausElement[4][1] = 1;
  ausElement[4][2] = 2;
  ausElement[5][0] = 5;
  ausElement[5][1] = 2;
  ausElement[5][2] = 3;
  ausElement[6][0] = 5;
  ausElement[6][1] = 3;
  ausElement[6][2] = 4;
  ausElement[7][0] = 5;
  ausElement[7][1] = 4;
  ausElement[7][2] = 1;

  usLastElement = 8;
  usLastNode = 6;
}

/*----------------------------*/
/* FUNCTION CreateIcosahedron */
/*----------------------------*/
static void CreateIcosahedron()
{
  FPTYPE fpDihedr = 138.1897222 / 180. * PI;
  FPTYPE fpSide = 4. / sqrt(2. * (5. + sqrt(5.)));
  FPTYPE fpHeight = fpSide * .5 * sqrt(3.);
  FPTYPE fpChord = sqrt(2. * fpHeight * fpHeight *
                   (1. - cos(fpDihedr)));
  FPTYPE fpTheta1 = PI - acos((2. - fpChord * fpChord) * .5);
  FPTYPE fpTheta2 = PI - fpTheta1;

  afpNode[0][0] = 0.0;
  afpNode[0][1] = 0.0;
  afpNode[0][2] = 1.0;

  afpNode[1][0] = sin(fpTheta1) * sin(0.0);
  afpNode[1][1] = sin(fpTheta1) * cos(0.0);
  afpNode[1][2] = cos(fpTheta1);

  afpNode[2][0] = sin(fpTheta1) * sin(2./5.*PI);
  afpNode[2][1] = sin(fpTheta1) * cos(2./5.*PI);
  afpNode[2][2] = cos(fpTheta1);

  afpNode[3][0] = sin(fpTheta1) * sin(4./5.*PI);
  afpNode[3][1] = sin(fpTheta1) * cos(4./5.*PI);
  afpNode[3][2] = cos(fpTheta1);

  afpNode[4][0] = sin(fpTheta1) * sin(6./5.*PI);
```

```
afpNode[4][1] = sin(fpTheta1) * cos(6./5.*PI);
afpNode[4][2] = cos(fpTheta1);

afpNode[5][0] = sin(fpTheta1) * sin(8./5.*PI);
afpNode[5][1] = sin(fpTheta1) * cos(8./5.*PI);
afpNode[5][2] = cos(fpTheta1);

afpNode[6][0] = 0.0;
afpNode[6][1] = 0.0;
afpNode[6][2] = -1.0;

afpNode[7][0] = sin(fpTheta2) * sin(1./5.*PI);
afpNode[7][1] = sin(fpTheta2) * cos(1./5.*PI);
afpNode[7][2] = cos(fpTheta2);

afpNode[8][0] = sin(fpTheta2) * sin(3./5.*PI);
afpNode[8][1] = sin(fpTheta2) * cos(3./5.*PI);
afpNode[8][2] = cos(fpTheta2);

afpNode[9][0] = sin(fpTheta2) * sin(PI);
afpNode[9][1] = sin(fpTheta2) * cos(PI);
afpNode[9][2] = cos(fpTheta2);

afpNode[10][0] = sin(fpTheta2) * sin(7./5.*PI);
afpNode[10][1] = sin(fpTheta2) * cos(7./5.*PI);
afpNode[10][2] = cos(fpTheta2);

afpNode[11][0] = sin(fpTheta2) * sin(9./5.*PI);
afpNode[11][1] = sin(fpTheta2) * cos(9./5.*PI);
afpNode[11][2] = cos(fpTheta2);

ausElement[0][0] = 0;
ausElement[0][1] = 1;
ausElement[0][2] = 2;

ausElement[1][0] = 0;
ausElement[1][1] = 2;
ausElement[1][2] = 3;

ausElement[2][0] = 0;
ausElement[2][1] = 3;
ausElement[2][2] = 4;

ausElement[3][0] = 0;
ausElement[3][1] = 4;
ausElement[3][2] = 5;

ausElement[4][0] = 0;
ausElement[4][1] = 5;
ausElement[4][2] = 1;
```

```
ausElement[5][0] = 1;
ausElement[5][1] = 2;
ausElement[5][2] = 7;

ausElement[6][0] = 2;
ausElement[6][1] = 3;
ausElement[6][2] = 8;

ausElement[7][0] = 3;
ausElement[7][1] = 4;
ausElement[7][2] = 9;

ausElement[8][0] = 4;
ausElement[8][1] = 5;
ausElement[8][2] = 10;

ausElement[9][0] = 5;
ausElement[9][1] = 1;
ausElement[9][2] = 11;

ausElement[10][0] = 1;
ausElement[10][1] = 7;
ausElement[10][2] = 11;

ausElement[11][0] = 2;
ausElement[11][1] = 7;
ausElement[11][2] = 8;

ausElement[12][0] = 3;
ausElement[12][1] = 8;
ausElement[12][2] = 9;

ausElement[13][0] = 4;
ausElement[13][1] = 9;
ausElement[13][2] = 10;

ausElement[14][0] = 5;
ausElement[14][1] = 10;
ausElement[14][2] = 11;

ausElement[15][0] = 6;
ausElement[15][1] = 7;
ausElement[15][2] = 8;

ausElement[16][0] = 6;
ausElement[16][1] = 8;
ausElement[16][2] = 9;

ausElement[17][0] = 6;
ausElement[17][1] = 9;
ausElement[17][2] = 10;
```

```
  ausElement[18][0] = 6;
  ausElement[18][1] = 10;
  ausElement[18][2] = 11;

  ausElement[19][0] = 6;
  ausElement[19][1] = 11;
  ausElement[19][2] = 7;

  usLastElement = 20;
  usLastNode = 12;
}

/*----------------------------*/
/* FUNCTION Tesselate         */
/*----------------------------*/
static void Tesselate()
{
  USHORT usLocalLastElement;
  USHORT usK;

  usLocalLastElement = usLastElement;
  for(usK = 0; usK < usLocalLastElement; usK++)
  {
    Bisector(usK, 0, 1);
    Bisector(usK, 1, 2);
    Bisector(usK, 2, 0);
    NewElements(usK);
  }
}

/*----------------------------*/
/* FUNCTION Bisector          */
/*----------------------------*/
static void Bisector(usK, usI, usJ)
USHORT usK;
USHORT usI;
USHORT usJ;
{
  USHORT usEndX1, usEndX2;
  FPTYPE fpNorm;
  FPTYPE fpE00, fpE01, fpE02, fpE10, fpE11, fpE12;
  FPTYPE afpX[3];

  usEndX1 = ausElement[usK][usI];
  usEndX2 = ausElement[usK][usJ];

  fpE00 = afpNode[usEndX1][0];
  fpE01 = afpNode[usEndX1][1];
  fpE02 = afpNode[usEndX1][2];
  fpE10 = afpNode[usEndX2][0];
```

```
  fpE11 = afpNode[usEndX2][1];
  fpE12 = afpNode[usEndX2][2];

  afpX[0] = fpE00 + fpE10;
  afpX[1] = fpE01 + fpE11;
  afpX[2] = fpE02 + fpE12;

  /* project new node onto surface */
  fpNorm = sqrt(afpX[0]*afpX[0] + afpX[1]*afpX[1] + afpX[2]*afpX[2]);
  afpX[0] /= fpNorm;
  afpX[1] /= fpNorm;
  afpX[2] /= fpNorm;

  afpNode[usLastNode][0] = afpX[0];
  afpNode[usLastNode][1] = afpX[1];
  afpNode[usLastNode][2] = afpX[2];
  ausNewNode[usI] = usLastNode;

  usLastNode++;
}

/*----------------------------*/
/* FUNCTION NewElements       */
/*----------------------------*/
static void NewElements(usK)
USHORT usK;
{
  USHORT usNewNodeX1 = ausNewNode[0];
  USHORT usNewNodeX2 = ausNewNode[1];
  USHORT usNewNodeX3 = ausNewNode[2];

  USHORT usOldNodeX1 = ausElement[usK][0];
  USHORT usOldNodeX2 = ausElement[usK][1];
  USHORT usOldNodeX3 = ausElement[usK][2];

  AddElement(usOldNodeX1, usNewNodeX1, usNewNodeX3);
  AddElement(usOldNodeX2, usNewNodeX1, usNewNodeX2);
  AddElement(usOldNodeX3, usNewNodeX2, usNewNodeX3);

  RenameElement(usK, usNewNodeX1, usNewNodeX2, usNewNodeX3);
}

/*----------------------------*/
/* FUNCTION AddElement        */
/*----------------------------*/
static void AddElement(usI, usJ, usK)
USHORT usI;
USHORT usJ;
USHORT usK;
{
  ausElement[usLastElement][0] = usI;
```

```
  ausElement[usLastElement][1] = usJ;
  ausElement[usLastElement][2] = usK;
  usLastElement++;
}

/*----------------------------*/
/* FUNCTION RenameElement     */
/*----------------------------*/
static void RenameElement(usK, usL, usM, usN)
USHORT usK;
USHORT usL;
USHORT usM;
USHORT usN;
{
  ausElement[usK][0] = usL;
  ausElement[usK][1] = usM;
  ausElement[usK][2] = usN;
}

/*----------------------------*/
/* FUNCTION GetElementAll     */
/*----------------------------*/
extern void GetElementAll(usK, pgeo)
USHORT usK;
PGEO pgeo;
{
  USHORT usNode1 = ausElement[usK][0];
  USHORT usNode2 = ausElement[usK][1];
  USHORT usNode3 = ausElement[usK][2];
  FPTYPE afpP1[3], afpP2[3], afpP3[3];
  FPTYPE afpU1[3], afpU2[3];
  FPTYPE afpX[3];
  FPTYPE afpN[3], fpDot, fpNorm;

  /* vertices */
  pgeo->afpP1[0] = afpP1[0] = afpNode[usNode1][0];
  pgeo->afpP1[1] = afpP1[1] = afpNode[usNode1][1];
  pgeo->afpP1[2] = afpP1[2] = afpNode[usNode1][2];
  pgeo->afpP2[0] = afpP2[0] = afpNode[usNode2][0];
  pgeo->afpP2[1] = afpP2[1] = afpNode[usNode2][1];
  pgeo->afpP2[2] = afpP2[2] = afpNode[usNode2][2];
  pgeo->afpP3[0] = afpP3[0] = afpNode[usNode3][0];
  pgeo->afpP3[1] = afpP3[1] = afpNode[usNode3][1];
  pgeo->afpP3[2] = afpP3[2] = afpNode[usNode3][2];

  /* vectors of edges */
  pgeo->afpU1[0] = afpU1[0] = afpP1[0] - afpP2[0];
  pgeo->afpU1[1] = afpU1[1] = afpP1[1] - afpP2[1];
  pgeo->afpU1[2] = afpU1[2] = afpP1[2] - afpP2[2];
  pgeo->afpU2[0] = afpU2[0] = afpP3[0] - afpP2[0];
  pgeo->afpU2[1] = afpU2[1] = afpP3[1] - afpP2[1];
```

```
  pgeo->afpU2[2] = afpU2[2] = afpP3[2] - afpP2[2];

  /* centroid */
  pgeo->afpX[0] = afpX[0] = (afpP1[0] + afpP2[0] + afpP3[0]) / 3.;
  pgeo->afpX[1] = afpX[1] = (afpP1[1] + afpP2[1] + afpP3[1]) / 3.;
  pgeo->afpX[2] = afpX[2] = (afpP1[2] + afpP2[2] + afpP3[2]) / 3.;

  /* jacobian, area, normal */
  afpN[0] = afpU1[1]*afpU2[2] - afpU1[2]*afpU2[1];
  afpN[1] = afpU1[2]*afpU2[0] - afpU1[0]*afpU2[2];
  afpN[2] = afpU1[0]*afpU2[1] - afpU1[1]*afpU2[0];
  pgeo->fpJacob = fpNorm = sqrt(afpN[0]*afpN[0] + afpN[1]*afpN[1] +
                                afpN[2]*afpN[2]);
  pgeo->fpDA = fpNorm * .5;
  afpN[0] /= fpNorm;
  afpN[1] /= fpNorm;
  afpN[2] /= fpNorm;
  fpDot = afpX[0]*afpN[0] + afpX[1]*afpN[1] + afpX[2]*afpN[2];
  if (fpDot < 0.)
  {
    pgeo->afpN[0] = -afpN[0];
    pgeo->afpN[1] = -afpN[1];
    pgeo->afpN[2] = -afpN[2];
  }
  else
  {
    pgeo->afpN[0] = afpN[0];
    pgeo->afpN[1] = afpN[1];
    pgeo->afpN[2] = afpN[2];
  }
}
```

B.5.3 C Header File my.h

```
/*----------------------------*/
/* my.h                       */
/*----------------------------*/
/* C header file with         */
/* general definitions        */
/*----------------------------*/
#ifdef SINGLEPREC
#  define FPTYPE float
#  define FPMASK "%f"
#else
#  define FPTYPE double
#  define FPMASK "%lf"
#endif
```

```
#define USHORT unsigned short
#define SHORT short
#define ULONG unsigned long
#define LONG long
#define INT long
#define BOOL unsigned short

#define PI 3.14159265358979323844
#ifndef RAND_MAX
#  define RAND_MAX 2147483648
#endif

#define max(a,b) ( ((a)>(b)) ? (a) : (b) )
#define min(a,b) ( ((a)<(b)) ? (a) : (b) )
#define SQR(x) ((x)*(x))
#define AABS(x) sqrt(SQR(x[0]) + SQR(x[1]) + SQR(x[2]))
#define DELTA(i,j) (((i) == (j)) ? 1.0 : 0.0)

#define FILENAM_SIZE 80
#define STRING_SIZE 80

#ifndef WWT
# ifndef XX_NUM_NODES
#   define XX_NUM_NODES 1
# endif
# ifndef G_MALLOC
#   define G_MALLOC(ulSize) malloc(ulSize);
# endif
#endif
```

B.5.4 C Header File geometry.h

```
/*---------------------------------*/
/* geometry.h                      */
/*---------------------------------*/
/* header file for geometry.c      */
/* and lap.c                       */
/*---------------------------------*/
/* programmed by Frank Traenkle    */
/*---------------------------------*/

/*-----------------------------*/
/* STRUCTs                     */
/*-----------------------------*/
typedef struct _GEO
{
  FPTYPE afpP1[3];   /* vertices */
```

```
  FPTYPE afpP2[3];
  FPTYPE afpP3[3];
  FPTYPE afpU1[3];   /* vectors of edges */
  FPTYPE afpU2[3];
  FPTYPE afpX[3];    /* centroid */
  FPTYPE afpN[3];    /* normal vector */
  FPTYPE fpDA;       /* area */
  FPTYPE fpJacob;    /* Jacobian */
} GEO, *PGEO;

/*----------------------------*/
/* PROTOTYPEs                 */
/*----------------------------*/
extern USHORT CreateGeometry();
extern void CloseGeometry();
extern void GetElementAll();
```

B.5.5 Make File lap.mak.include

```
#-------------------------------------------------------
# lap.mak.include
#-------------------------------------------------------
# included general file into makefiles for different
# protocols
#-------------------------------------------------------

.KEEP_STATE:

# Programming model used.
MODEL = parmacs

# gcc
CC = gcc

# Target name
TARGET = lap
TARGETDIR = .

# Invariants, you shouldn't (typically) override these.
INC = -I$(WWT_ROOT)/Include/$(PROTOCOL) -I$(WWT_ROOT)/Include

STD_CFLAGS = $(INC)
STD_LDFLAGS = -n -dc -dp -e __wwt_startup_ -X

# You should not change these.
CFLAGS = $(STD_CFLAGS) $(OTHER_CFLAGS)
LDFLAGS = $(STD_LDFLAGS) $(OTHER_LDFLAGS)
```

```
WWTLIBDIR = $(WWT_ROOT)/Lib
#LIBDIRS = -L$(WWTLIBDIR) -L$(WWTLIBDIR)/$(PROTOCOL)
#LIBS =  -l$(MODEL) -lm -lc
# gcc
LIBDIRS = -L$(WWTLIBDIR) -L$(WWTLIBDIR)/$(PROTOCOL) \
         -L/usr/local/lib/gcc-lib/sparc-sun-sunos4.1.2/2.4.5
LIBS =  -l$(MODEL) -lm -lgcc -lc

# /lib/crt0.o provides a pointer to the environment needed by some
# lib objects
OBJS = lap.o geometry.o /lib/crt0.o

# PARMACS stuff
MACDIR = $(WWTLIBDIR)
MACROS = $(MACDIR)/c.m4.local $(MACDIR)/c.m4.monmacs $(MACDIR)/c.m4.smacs

.SUFFIXES:
.SUFFIXES: .o .c .U .h .H
.c.o: ; $(CC) -c $(CFLAGS) $*.c
.U.c: ; m4 $(MACROS) $*.U >$*.c
.H.h: ; m4 $(MACROS) $*.H >$*.h

$(TARGET): $(OBJS)
ld $(LDFLAGS) $(OBJS) -o $(TARGET).no-vt $(LIBDIRS) $(LIBS)
vt -a $(TARGET) -d -w $(OTHER_VTFLAGS) $(TARGET).no-vt
#       wwt_strip $(TARGET)
compress $(TARGET)

rmtarget:
rm -f $(TARGETDIR)/$(TARGET) $(TARGETDIR)/$(TARGET).Z

install: $(TARGET)
cp $(TARGET).Z $(TARGETDIR)

# file dependencies.
lap.o: lap.c geometry.h
geometry.o: geometry.c geometry.h

# "clean" rule.
clean:
/bin/rm -f $(TARGET) $(TARGET).Z $(TARGET).no-vt lap.o geometry.o
```

B.5.6 Make File Makefile

```
PROTOCOL = dir1SW
```

```
# OTHER_CFLAGS = -O2 -DWWT -DPAR_INIT -DDO_PRINT -DDIR1 -DSIZE=1026
OTHER_CFLAGS = -Wall -O2 -DWWT -DDIR1 -DCICO -DTIMERS -DSINGLEPREC

OTHER_LDFLAGS =

OTHER_VTFLAGS =

include lap.mak.include
```

Bibliography

[1] N. Amann. Mobility Problems in Bounded Stokes Flow by Boundary Integral Methods. Master's thesis, University of Wisconsin – Madison, 1992.

[2] N. Amann and S. Kim. Parallel Computational Microhydrodynamics: Scalable Load-Balancing Strategies. *Engineering Analysis with Boundary Elements*, 11:269–276, 1993.

[3] G.M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conference Proceedings*, pages 483–485, April 1967.

[4] G.K. Batchelor. Transport Properties of Two-Phase Materials with Random Structure. *Ann. Rev. Fluid Mech.*, 6:227–255, 1974.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation - Numerical Methods.* Prentice-Hall, 1989.

[6] R.B. Bird, W.E. Stewart, and E.N. Lightfoot. *Transport Phenomena.* Wiley, New York, 1960.

[7] C.Y. Chan, A.N. Beris, and S.G. Advani. Second-order Boundary Element Method Calculations of Hydrodynamic Interactions between Particles in Close Proximity. *Intl. J. Numer. Meth. Fluids*, 14:1063–1087, 1992.

[8] W.C. Chew and P.N. Sen. Dielectric Enhancement due to Electrochemical Double Layer: Thin Double Layer Approximation. *J. Chem. Phys.*, 77(9), 1992.

[9] E.G. Coffman and G.S. Lueker. *Probalistic Analysis of Packing and Partitioning Algorithms.* Wiley, New York, 1991.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Toward a Realistic Model of Parallel Computation. *In Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–12, May 1993.

[11] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. *Introduction to Split-C Version 1.0.* University of California, Berkeley, 1993.

[12] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. *In Proceedings of Supercomputing 1993*, 1993.

[13] T.von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *19th Ann. Symp. on Computer Architecture*, pages 256–266, 1992.

[14] M.J. Flynn. Very High–speed Computers. *Proc. IEEE*, 54:1901–1909, 1966.

[15] A. Friedman. *Foundations of Modern Analysis.* Dover, New York, 1982.

[16] Y.O. Fuentes. *Parallel Computational Strategies for Multiparticle Systems in Stokes Flow.* PhD thesis, University of Wisconsin – Madison, 1990.

[17] Y.O. Fuentes and S. Kim. Parallel Computational Microhydrodynamics: Communication Scheduling Strategies. *A.I.Ch.E. Journal*, 38:1059–1078, 1992.

[18] Y.O. Fuentes, S. Kim, and D.J. Jeffrey. Mobility Functions for Two Unequal Viscous Drops in Stokes Flow. I. Axisymmetric Motions. *Phys. Fluids*, 31(9):2445–2455, 1988.

[19] Y.O. Fuentes, S. Kim, and D.J. Jeffrey. Mobility Functions for Two Unequal Viscous Drops in Stokes Flow. II. Asymmetric Motions. *Phys. Fluids A*, 1(1):61–76, 1989.

[20] A. Gerstlauer. Boundary Integral Equation Methods for Thin Particles in Stokes Flow. Master's thesis, University of Wisconsin – Madison, 1989.

[21] G.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17:263–269, 1969.

[22] S.R. Graubard. *A New Era in Computation.* DAEDALUS J. Amer. Acad. Arts and Sciences, Winter 1992.

[23] M.D. Hill and J.R. Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the A.C.M.*, 33(8):97–102, 1990.

[24] M.D. Hill, J.R. Larus, S.K. Reinhardt, and D.A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, To appear, November 1993.

[25] J.D. Jackson. *Classical Electrodynamics.* Wiley, New York, 1975.

[26] D.J. Jeffrey. Conduction through a Random Suspension of Spheres. *Proc.Roy.Soc.Lond.*, A 335(355), 1973.

[27] S.J. Karrila. *Linear Operator Theory Applied to Fast Computational Strategies for Particle Interactions in Viscous Flows.* PhD thesis, University of Wisconsin – Madison, 1988.

[28] S.J. Karrila, Y.O. Fuentes, and S. Kim. Parallel Computational Strategies for Hydrodynamic Interactions between Rigid Particles of Arbitrary Shape in a Viscous Fluid. *J. Rheology*, 33:913–947, 1989.

[29] S.J. Karrila and S. Kim. Integral Equations of the Second Kind for Stokes Flow: Direct Solution for Physical Variables and Removal of Inherent Accuracy Limitations. *Chem. Eng. Comm.*, 89:123–161, 1989.

[30] Kendall Square Research. *Kendall Square Research Technical Summary*, 1992.

[31] S. Kim and S.J. Karrila. *Microhydrodynamics: Principles and Selected Applications.* Butterworth–Heinemann, Boston, 1991.

[32] D.J. Klingenberg, F. van Swol, and C.F. Zukoski. Dynamic Simulation of Electrorheological Suspensions. *J. Chem. Phys.*, 91:7888–7895, 1989.

[33] D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[34] C. Lin and L. Snyder. A Comparison of Programming Models for Shared Memory Multiprocessors. *In Proceedings of the 1990 International Conference on Parallel Processing*, 2:163–170, August 1990.

[35] S.Y. Lu. *On the Effective Transport Properties of Composite Materials.* PhD thesis, University of Wisconsin – Madison, 1988.

[36] S.Y. Lu and S. Kim. Effective Thermal Conductivity of Composites Containing Spheroidal Inclusions. *A.I.Ch.E. Journal*, 36(6), 1990.

[37] L.E. Malvern. *Introduction to the Mechanics of a Continuous Medium.* Prentice-Hall, 1969.

[38] C. Maul. Boundary Integral Equation Methods for Polyhedra in Stokes Flow. Master's thesis, University of Wisconsin – Madison, 1990.

[39] J. Maxwell. *Electricity and Magnetism.* Dover, 1954.

[40] C.W. Oseen. *Hydrodynamik.* Akad. Verlagsgesellschaft, Leipzig, 1927.

[41] P. Pakdel and S. Kim. Mobility and Stresslet Functions of Particles with Rough Surfaces in Viscous Fluids: A Numerical Study. *J. Rheol.*, 11:797–823, 1991.

[42] N. Phan-Thien and S. Kim. Microstructures in Elastic Media. To appear, 1994.

[43] H. Power and G. Miranda. Second Kind Integral Equation Formulation of Stokes' Flows past a Particle of Arbitrary Shape. *SIAM J. Appl. Math.*, 47(4):689–698, 1987.

[44] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C.* Cambridge, New York, 2nd edition, 1992.

[45] D. Ramkrishna and N.R. Amundson. *Linear Operator Methods in Chemical Engineering.* Prentice-Hall, Englewood Cliffs, 1985.

[46] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *In Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[47] A.H. Stroud and D. Secrest. *Gaussian Quadrature Formulas.* Prentice-Hall, Englewood Cliffs, 1966.

[48] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[49] Thinking Machines Corporation. *CMMD User's Guide Version 3.0*, 1993.

[50] F. Traenkle, M.I. Frank, M.K. Vernon, and S. Kim. Solving Microstructure Electrostatics with MIMD Parallel Supercomputers and Split-C. To appear in J. Non-Newtonian Fluid Mech., 1993.

[51] F. Traenkle, M.D. Hill, and S. Kim. Solving Microstructure Electrostatics on a Proposed Parallel Computer. To appear in Computers and Chemical Engineering, 1993.

[52] S. Wolfram. *Mathematica.* Addison-Wesley, 2nd edition, 1991.

[53] M. Zuzovsky and H. Brenner. Effective Conductivities of Composite Materials Composed of Cubic Arrangements of Spherical Particles Embedded in an Isotropic Matrix. *J. Appl. Math. Phys.*, 28(6), 1977.