Solving Microstructure Electrostatics on a Proposed Parallel Computer*

Frank Traenkle[1], Mark D. Hill[2] and Sangtae Kim[1,2]
[1]Department of Chemical Engineering
[2]Computer Sciences Department
University of Wisconsin–Madison, Madison, WI 53706, U.S.A.

April 6, 1994

Abstract

The programming models presented by parallel computers are diverse and changing. We study a new parallel programming model—cooperative shared memory (CSM)—with a collaborative effort between chemical engineers and computer scientists. Since CSM machines do not (yet) exist we evaluate our applications and machine designs with the Wisconsin Wind Tunnel (WWT), which runs CSM programs and calculates the performance of hypothetical parallel computers.

The application considered is the class of three–dimensional elliptic partial differential equations (Laplace, Stokes, Navier) with solutions represented by boundary integral equations. The parallel algorithm follows naturally from our use of the Completed Double Layer Boundary Integral Equation Method (CDLBIEM).

A major result is the demonstration that coding CDLBIEM is much simpler under CSM than with the message passing model, and yet performance (computational times and speed ups) is comparable, a fact that may be of great interest to designers of future machines. With WWT, we can also examine performance as a function of machine parameters such as cache size and network bandwidth and latency. The possibility of tweaking simultaneously the algorithm and architecture to outline pathways of evolution for future parallel machines is an important concept explored in this work.

Keywords: electrostatics, Laplace equation, potential theory;
boundary element, boundary integral equation;
cache coherence, $\mathrm{DIR_1SW}$, shared memory.

*To appear in *Computers and Chemical Engineering* special issue on parallel computing.

1 Introduction

Economic factors are forcing the convergence of parallel computer hardware to a collection of workstation-like nodes connected by a fast, usually custom network [Culler et al., 1993] (see Figure 1). A similar consensus, however, has not developed for the mechanisms this hardware should provide to support the communication and synchronization necessary to execute parallel programs. In large part, this situation is due to the (temporary or permanent) absense of a generally-accepted parallel programming model.

Parallel programming models are an abstraction of parallel machines that allow users to design parallel programs. To be effective the models must be both natural to use and straightforward to implement.

This present effort brings users (chemical engineers) and implementors (computer scientists) together to evaluate and refine a new parallel programming model called CSM (Cooperative Shared Memory). CSM allows users to view all their data as residing in a common memory but asks them to manage performance with data buffering hints [Hill et al., 1993]. Since no CSM machines exist we evaluate our designs with the Wisconsin Wind Tunnel (WWT) [Reinhardt et al., 1993]. WWT runs CSM programs and calculates expected performance of a specific implementation. WWT is key to this research as it lets us use a machine that does not exist to influence what the next generation of parallel machines would look like.

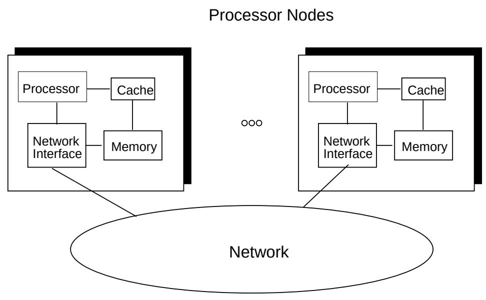

Figure 1: Organization of a parallel computer. Many parallel computers are composed of P processor nodes, each of which contains a microprocessor, cache, and physically local memory. The nodes are connected by an interconnection network (e.g., a 2D mesh or fat tree).

Elliptic partial differential equations (PDEs) provide an excellent vehicle for trying out these ideas. They are encountered in diverse phenomena in science and engineering: flow through porous media, steady-state heat transfer, and very viscous flows are some of the chemical engineering examples that come to mind. A boundary integral equation approach appropriate for complex geometries associated with structured media [Kim and Karrila, 1991; Phan-Thien and Kim, 1994] has successfully tackled these elliptic PDEs on a number of parallel computers [Amann and Kim, 1993; Fuentes and Kim, 1992; Karrila *et al.* 1989; Traenkle *et al.* 1993], and presents itself as a logical test algorithm for programming models and proposed parallel computers.

Our first results in this direction are reported here. A boundary integral solution of the three dimensional Laplace equation governing the interaction between charged particles in a dielectric has been implemented in CSM. Our results indicate that for this class of boundary integral computations, CSM using a simple directory cache–coherence protocol (Dir_1SW, see Section 5.1) scales very effectively to many processors (we obtained a 7.74-fold speedup from 16 to 128 processors for our problem). Our experimental measurements have gone up to simulations of 256 processors;

with access to larger machines to host the simulations, we plan to extend these shared–memory experiments to over 1000 processors. This is an important finding given the generally prevalent view that shared memory does not scale effectively to large numbers of processors. Even though we have previously validated the Wisconsin Wind Tunnel against an existing system [Reinhardt et al., 1993], projections to any non-existent system may still contain errors [Rettberg and Thomas, 1986].

2 Programming Models for Parallel Computers

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. 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.* We will discuss the first two but not data parallel, because data parallel's generality is not (yet) established.

The message-passing programming model exposes the hardware of a P-processor computer and asks the programmer or compiler to write a program with the P processors, that use P address spaces, and communicate with explicit messages. An *address space* is a mapping from addresses — conceptually variable names — to memory locations — conceptually storage for variable values. Thus, on a message-passing computer, the hardware will treat accesses to variable `X` by two different processors as referring to two different locations. If `X` is to refer to the same location — as in a language like HPF (high performance Fortran) — the compiler and runtime system must send messages to create the illusion. The cost model for message-passing is that messages have large start-up costs, implying that performance will be better for programs that can 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.

The term "shared memory" may suggest that all memory resides 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 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.

One drawback of hiding the location of data is that it obscures the real cost of a shared-memory access leading some to argue that the appropriate cost model for shared memory is that all memory accesses have equal cost [Lin and Snyder, 1990]. Furthermore, hardware designers often use the caches in each node to keep copies of some memory locations so that subsequent accesses can be serviced quickly (e.g., Kendall Square KSR-1 [Kendall Square Research, 1992] and Stanford DASH [Lenoski et al., 1992]). These caches usually hold the most recently accessed locations since empirical evidence shows that this data is most likely to be accessed next. Cached copies of data differ from software created copies, because they retain their original address and–in a technique called *cache coherence*–they will be automatically reclaimed by hardware if another processer writes the original memory location. A second difference is that caches store copies in fixed-sized *blocks* of 32 to 128 bytes (4-16 double-precision floating-point numbers), while software copies can be of any size.

The next section introduces a more realistic cost model for programming on cache-coherent shared-memory computers.

3 Cooperative Shared Memory

Cooperative Shared Memory is a programming model that relies on shared-memory semantics for correctness, but then provides a cost model and performance primitives that allow the programmer to understand and manage program performance on cache-coherent shared-memory hardware [Hill et al., 1993]. The model is "cooperative" in two ways. First, it discourages programs whose processors compete when accessing shared data, and instead favors that they coordinate to move data as little as possible. Second, it identifies sharing patterns that hardware can support effectively to encourage their use; thus, allowing the software and hardware to cooperate.

Our initial implementation of cooperative shared memory, called *Check-In/Check-Out (CICO)*, asks the programmer to think of the computer as having a cache at each processor but not to worry about the physical location of memory. We then ask the programmer to move data between cache and memory with three annotations. However, unlike message-passing, the annotations are "hints" that may be omitted or inserted aggressively without affecting the correctness of the program. The final paragraph of this section discusses the advantages of this decoupling of performance and correctness.

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 the check-out expects to be the only processor accessing the data until it is checked-in. `check_out_S` asserts that the processor is willing to share read-only access to the block. `check_in` marks the end of an interval in which a processor uses the data. Operationally, the `check_out` annotations can be viewed as fetching a copy of the data into the processor's cache, as if the processor directly accessed the block. In contrast, the `check_in` annotation can be viewed as flushing the data from the processor's cache back to memory, as if the processor's cache had replaced it.

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 (e.g., the programmer expects exclusive access and multiple processes access the same data). Third, communication occurs when processors require exclusive access to different variables that happen to be placed in the same cache block, called *false sharing*.

Currently, we ask programmers to `check_out` and `check_in` cache blocks, but we provide tools that give feedback on what variable accesses cause communication so that programmers can "performance debug" their programs. In the future, we hope to associate CICO annotations with high-level language data types so that the compiler can manage both large data aggregates and eliminate false sharing. Nevertheless, we expect the programmer will still need to manage the big picture of communication.

Since the purpose of CICO annotations allow programmers to identify communication, why not just have programmer use message passing, where communication is explicit? The key advantage of

CICO is that function and performance are separate. Adding CICO annotations can never break a correct program. Thus, the optimizations can be added aggressively to important parts of program and be completely omitted in rarely-executing parts. Furthermore, since CICO annotations do not change a datum's address, a program can optimize one routine without changing all routines that interact with it. With message-passing, on the other hand, performance optimizations must be done carefully to avoid transformations that could introduce functional bugs.

4 Boundary Integral Equation Methods for Laplace's Equation

In this section we derive boundary integral equations (BIEs) that model the electric potential generated by N fixed charged bodies. These boundary integral equations are equivalent to the differential Laplace equation but have the advantage that the 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 specific version used here, the Completed Double Layer Boundary Integral Equation Method (CDLBIEM), is amenable to solution by fixed-point iterative methods.

We consider two problems:

1. The N bodies are located at fixed points in an unbounded infinite void domain.

2. The N bodies are inclusions placed inside a spherical perfect conductor.

The standard CDLBIEM approach for problem 1 is readily extended to handle problem 2, using the method of images. The bounded domain problem, while providing additional physical insight, also introduces a well characterized increase in the computation to communication ratio — there is an increase in the computational load per processor, without a corresponding increase in inter–processor communications. The final equations for both arrays are of the same structure and are solved by the same algorithm as described in the third part of this section. Since the load balance has a large impact on the program's scaling, we divide the workload onto the processors according to a heuristic algorithm called *Largest Processing Time first* [Graham, 1969] and described in Section 4.4. Finally special issues of CDLBIEM's implementation in the Cooperative Shared Memory (CSM) model are discussed.

4.1 N Bodies in an Unbounded Domain

The boundary integral approach is particularly well suited to handle the electrostatic interactions between N bodies of *arbitrary shape* placed at specified locations in a 3-dimensional void[†] space of infinite extent. However, for purposes of demonstrating the basic method, we present results for the special case of N spheres. The assumption that they are perfect conductors ensures that their surfaces are equipotentials. The derivation is presented in detail for the single-particle geometry, but the extension to the multi-particle geometry is quite straightforward as summarized in the subsequent section.

[†]The analysis is readily extended to a dielectric medium by introducing the dielectric permittivity.

4.1.1 CDLBIEM for One Body

The electric potential $\psi(\boldsymbol{x})$ in the void infinite domain satisfies the Laplace equation and the following boundary condition:

$$\begin{array}{lll} \text{PDE} & \nabla^2\psi = 0 & ;\boldsymbol{x} \in V^0 \\ \text{BC} & \psi\big|_{|\boldsymbol{x}|\to\infty} = \psi^\infty(\boldsymbol{x}) & \end{array} \quad . \tag{1}$$

Here V^0 denotes the open infinite 3-dimensional domain exterior to the body and $S = \partial V$ is the surface of the body. In the case of a uniform applied (macroscopic) electric field $\boldsymbol{E}^\infty$ the ambient field can be written explicitly as $\psi^\infty = -\boldsymbol{E}^\infty \cdot \boldsymbol{x}$.

To derive the boundary integral representation we start with the basic relations: the Green's function or the fundamental solution of the Laplace equation; Green's identity; and the integral representation. The proofs are readily available in a number of references on potential theory, e.g., [Jackson, 1975]. The classical integral representation is then modified to yield the *completed double layer representation* which we use in our work.

The Green's function for the Laplace equation is

$$G(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} , \tag{2}$$

and satisfies the following PDE and homogeneous boundary condition:

$$\begin{array}{llll} \text{PDE} & \nabla^2_{\xi} G = -\delta(\boldsymbol{\xi} - \boldsymbol{x}) & ; & \boldsymbol{\xi} \in V^0 \\ \text{BC} & G\big|_{|\boldsymbol{\xi}|\to\infty} = 0 \; . & & \end{array} \tag{3}$$

This Green's function is used with Green's theorem (Green's second identity),

$$\int_V \left(\nabla^2_{\xi} G\psi(\boldsymbol{\xi}) - G\nabla^2_{\xi}\psi(\boldsymbol{\xi})\right) dV_{\xi} = \oint_{S+S_\infty} (\nabla_{\xi} G\psi - G\nabla_{\xi}\psi) \cdot \boldsymbol{n}(\boldsymbol{\xi})\, dS_{\xi} \; , \tag{4}$$

and the boundary condition of (1) at infinity to arrive at the integral representation,

$$\left.\begin{array}{ll} \boldsymbol{x} \text{ inside of } V & \psi(\boldsymbol{x}) \\ \boldsymbol{x} \text{ outside of } V & 0 \\ \boldsymbol{x} \text{ on } S = \partial V & \frac{1}{2}\psi(\boldsymbol{x}) \end{array}\right\} = \psi^\infty(\boldsymbol{x}) - \oint_S G(\boldsymbol{x}, \boldsymbol{\xi})\, \hat{\boldsymbol{n}}(\boldsymbol{\xi}) \cdot \nabla_{\xi}\psi(\boldsymbol{\xi})\, dS_{\xi} + \frac{1}{4\pi} \oint_S \frac{\hat{\boldsymbol{n}}(\boldsymbol{\xi}) \cdot (\boldsymbol{x} - \boldsymbol{\xi})}{|\boldsymbol{x} - \boldsymbol{\xi}|^3} \psi(\boldsymbol{\xi})\, dS_{\xi} \; . \tag{5}$$

The normal vector $\hat{\boldsymbol{n}} = -\boldsymbol{n}$ points out of the body into V. The first integral on the RHS, the so called *single layer potential*, corresponds physically to a field generated by a surface distribution of charges. The second integral, the *double layer potential*, is a field generated by a surface distribution of electric dipoles. It will prove convenient to define

$$K(\boldsymbol{x}, \boldsymbol{\xi}) \;=\; \frac{(\boldsymbol{x} - \boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi})}{2\pi|\boldsymbol{x} - \boldsymbol{\xi}|^3} \tag{6}$$

as the entity for the kernel of the double layer operator.

The classical integral representation (5) can be used directly as a computational method; given the potential on the surface, we solve the integral equation for the unknown surface charge density which is proportional to $\hat{\boldsymbol{n}} \cdot \nabla_{\xi}\psi(\boldsymbol{\xi})$. However, a far more efficient computational method based on the double layer potential can be derived as we now show.

The double layer part of the integral representation may be rewritten as

$$\mathcal{K}\varphi(\boldsymbol{\eta}) = \oint_S K(\boldsymbol{x}, \boldsymbol{\xi})\, \varphi(\boldsymbol{\xi})\, dS_{\xi} \ ,$$

where $\varphi(\boldsymbol{\eta}) = \frac{1}{2}\psi(\boldsymbol{\eta})$, $\boldsymbol{\eta} \in S$ is the *double layer density* on the sphere surface and $\mathcal{K}$ is the *double layer operator.* Because of the weak singularity (as $\boldsymbol{x} \to \boldsymbol{\xi}$) in $K(\boldsymbol{x}, \boldsymbol{\xi})$, the double layer kernel is a compact linear operator [Friedman, 1982].

An essential property of the linear double layer operator are its jump conditions on the body surface. As the point of evaluation, $\boldsymbol{x}$ goes from outside to S, and from S to inside, the jumps in Equation 5 of $\psi(\boldsymbol{\eta})/2$ are due solely to the double layer potential because the single layer potential is obviously continuous across S. Indeed, for any double layer density φ, we can show from first principles that given $\boldsymbol{x} = \boldsymbol{\eta} + \epsilon\hat{\boldsymbol{n}}$ (so that for $\epsilon > 0$, $\boldsymbol{x}$ is inside V; for $\epsilon < 0$, $\boldsymbol{x}$ is outside V; and for $\epsilon = 0$, $\boldsymbol{x} = \boldsymbol{\eta}$ is on S) the jump properties of the double layer potential may be written explicitly as:

$$\lim_{\epsilon \to 0+} \oint_S K(\boldsymbol{x}, \boldsymbol{\xi})\, \varphi(\boldsymbol{\xi})\, dS_{\xi} = \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi})\, \varphi(\boldsymbol{\xi})\, dS_{\xi} + \varphi(\boldsymbol{\eta}) \tag{7}$$

$$\lim_{\epsilon \to 0-} \oint_S K(\boldsymbol{x}, \boldsymbol{\xi})\, \varphi(\boldsymbol{\xi})\, dS_{\xi} = \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi})\, \varphi(\boldsymbol{\xi})\, dS_{\xi} - \varphi(\boldsymbol{\eta}) \ . \tag{8}$$

It is evident that the double layer potential, as a superposition of dipole fields, cannot represent the field produced by a charged conductor. That is precisely the role of the single layer potential. However, in place of the single layer potential, we can use the field generated by a point charge, with charge equal to the known net charge Q_α on particle α and position $\boldsymbol{x}_\alpha$ at any arbitrary point inside the body. This yields the *completed double layer boundary integral representation*,

$$\psi(\boldsymbol{x}) = \psi^{\infty}(\boldsymbol{x}) + \frac{Q_\alpha}{|\boldsymbol{x} - \boldsymbol{x}_\alpha|} + \oint_{S_\alpha} K(\boldsymbol{x}, \boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\xi} \qquad ; \boldsymbol{x} \in V^0 \ . \tag{9}$$

On the surface of the particle, we employ the boundary condition and formally substitute the surface potential $\psi(\boldsymbol{\eta})$ by the constant unknown ψ_α to get the boundary integral equation. Now because of the jump condition (7), the boundary integral equation for the unknown potential ψ_α on the body surface is

$$\psi_\alpha = \psi(\boldsymbol{\eta}) = \psi^{\infty}(\boldsymbol{\eta}) + \frac{Q_\alpha}{|\boldsymbol{\eta} - \boldsymbol{x}_\alpha|} + \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi})\varphi(\boldsymbol{\xi})\, dS_{\xi} + \varphi(\boldsymbol{\eta}) \quad ; \boldsymbol{\eta} \in S \ , \tag{10}$$

note that the operator that acts on $\varphi(\boldsymbol{\eta})$ is $1 + \mathcal{K}$.

Since the complete potential cannot be created by the double layer operator itself, the range of the operator $1 + \mathcal{K}$ is incomplete: the orthogonal complement of its range has dimension one. By the Fredholm theorems, $N(1 + \mathcal{K})$, the null space of $1 + \mathcal{K}$, must also have dimension one [Friedman, 1982]. In fact, the constant function $\varphi = 1$ on the surface of the particle is the basis of $N(1 + \mathcal{K})$. Since nonzero surface potentials and charges go hand in hand, we may complete the integral representation by assigning the unknown potential to a projection of the double layer density on the null space,

$$\psi_\alpha = -\frac{1}{S_\alpha} \oint_{S_\alpha} \varphi\, dS = -\varphi^{(\alpha)} \langle \varphi, \varphi^{(\alpha)} \rangle \ . \tag{11}$$

Here S_α denotes the surface area of particle α and

$$\varphi^{(\alpha)} = 1/\sqrt{S_\alpha} \tag{12}$$

is the normalized basis of $N(1+\mathcal{K})$ with respect to the inner product,

$$\langle f, g \rangle = \oint_{S_\alpha} f g \, dS \ . \tag{13}$$

Equation (11) is a further condition for the double layer density $\varphi^{(\alpha)}$ and is inserted into Equation (10). The resulting equation for the unknown double layer density on the particle surface S_α is:

$$\varphi(\boldsymbol{\eta}) = -\psi^\infty(\boldsymbol{\eta}) - \frac{Q_\alpha}{|\boldsymbol{\eta} - \boldsymbol{x}_\alpha|} - \frac{1}{S_\alpha} \oint_{S_\alpha} \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} - \oint_S K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \ . \tag{14}$$

In Equation (14) the operator $\mathcal{K}$ is replaced by the new operator $\mathcal{H} = \mathcal{K} + \varphi^{(\alpha)} \langle \bullet, \varphi^{(\alpha)} \rangle$ that can be viewed as the *Wielandt deflated* double layer operator. The eigenvalue of $\mathcal{K}$ with the largest norm is an isolated eigenvalue at -1 [Friedman, 1982] and the Wielandt deflation shifts it to the origin. Consequently, the spectral radius of $\mathcal{H}$ is less than one and $\mathcal{H}$ is a contraction map. We have thus arrived at the major purpose of the completed double layer representation: a well posed *Fredholm integral equation of the second kind* that is amenable to iterative solution to the unique fixed point. In subsequent sections, we will show that such iterative solutions fit naturally onto parallel computers. We conclude this section with two items: the extension of the representation to the N–body problem and the numerical approach to solve the boundary integral equation.

4.1.2 CDLBIEM for N Bodies

The equations derived so far are valid for just one body α in an infinite void space. In this section the equations are generalized to describe N bodies in an infinite void 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, thus 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* (9) is the union of all body surfaces S_k. Instead of having just one point charge, we have an array of N point charges that create a sum of potential fields. These ideas are expressed in the following equation corresponding to the equation for one body (9):

$$\psi(\boldsymbol{x}) = \psi^\infty(\boldsymbol{x}) + \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{x} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{x}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \right\} \quad ; \boldsymbol{x} \in V^0 \ . \tag{15}$$

Using the jump condition (7) we obtain the boundary integral equation on the surface S_k of particle k corresponding to Equation (10):

$$\psi_k = \psi(\boldsymbol{\eta}) = \psi^\infty(\boldsymbol{\eta}) + \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{\eta} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \right\} + \varphi(\boldsymbol{\eta}) \quad ; \boldsymbol{\eta} \in S_k \ . \tag{16}$$

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

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

$$\varphi^{(k)}(\boldsymbol{x}) = \begin{cases} \frac{1}{\sqrt{S_k}} & ; \boldsymbol{x} \in S_k \\ 0 & ; \boldsymbol{x} \notin S_k \end{cases} \qquad ; k = 1 \ldots N \; . \tag{17}$$

After Wielandt's deflation has been applied in parallel for all basis functions of the nullspace, the following integral equations for the unknown double layer density φ are obtained:

$$\varphi(\boldsymbol{\eta}) = -\psi^{\infty}(\boldsymbol{\eta}) - \frac{1}{S_k} \oint_{S_k} \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} - \sum_{l=1}^{N} \left\{ \frac{Q_l}{|\boldsymbol{\eta} - \boldsymbol{x}_l|} + \oint_{S_l} K(\boldsymbol{\eta}, \boldsymbol{\xi}) \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \right\} \tag{18}$$

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

There is one equation for each body surface S_k. This system of integral equations can not 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 iterative methods. Having obtained the solution φ of the equation system (18), the potential on each body k can be calculated using Equation (11):

$$\psi_k = -\frac{1}{S_k} \oint_{S_k} \varphi(\boldsymbol{\xi}) \, dS_{\boldsymbol{\xi}} \; . \tag{19}$$

The potential in the vacuum is readily available by evaluating Equation (15).

4.1.3 Numerics

This section describes the numerical approach to solve Equation (18). The numerical solution scheme boils down to (1) discretization of the particle surfaces into boundary elements; (2) approximation of the double layer densities by a basis set; and (3) approximation of the integrals by quadratures. The integral equation then reduces to a large linear system in which the unknown vector is the discretized representation of the double layer density.

Each body surface is discretized into M boundary elements, which are planar triangles. Those are sufficient for the considered case of spherical bodies. First order collocation is used to reduce the continuous domain of the unknown double layer density $\varphi(\boldsymbol{\eta})$ to a finite set of variables at distinct collocation points. In our approach to the problem, one boundary element has one collocation point, its centroid, and $\varphi(\boldsymbol{\eta})$ is assumed to be constant on the whole boundary element. However, to solve problems with more complex body shapes, curved elements, higher order interpolation and higher order quadratures will be especially useful [Chan et al., 1992].

The double layer integral occuring in the boundary integral equation (18) is approximated by Gaussian-Legendre quadrature. The common quadrature formulas for 1-dimensional domains are generalized to the case of 2-dimensional boundary elements: for each natural coordinate direction a 1-dimensional Gaussian-Legendre quadrature is performed.

The resulting numerical approximation of the boundary integral equations (18) is:

$$\varphi_{ki} = \varphi(\boldsymbol{\eta}_{ki}) = \underbrace{-\psi^{\infty}(\boldsymbol{\eta}_{ki}) - \sum_{l=1}^{N} \frac{Q_l}{|\boldsymbol{\eta}_{ki} - \boldsymbol{x}_l|}}_{=b_{ki}} \tag{20}$$

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

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

where

- k denotes the kth body and i its ith boundary element.
- $\boldsymbol{\eta}_{ki}$ is the collocation point on this boundary element.
- S_{ki} is its area.
- $\boldsymbol{p}_{ki,1}$ is one of its vertices and the vectors $\boldsymbol{u}_{ki,1}$ and $\boldsymbol{u}_{ki,2}$ represent its edges.
- w_r and $\tilde{w}_{rs}$ are the weights, x_r and $\tilde{x}_{rs}$ the Q collocation points of the quadrature.

The discretized boundary integral equation (20) is a system of linear algebraic equations, which can be written as

$$\boldsymbol{x} = \tilde{\boldsymbol{A}} \cdot \boldsymbol{x} + \tilde{\boldsymbol{b}} \ . \tag{21}$$

The unknown vector $\boldsymbol{x}$ contains the collocated double layer density as its elements: $x_{(k-1)M+i} = \varphi_{ki}$. The largest eigenvalues of the system matrix $\tilde{\boldsymbol{A}}$ are quite close to those of $\mathcal{H}$. Thus parallel iterative methods are applied to solve this system, as described in the next sections.

4.2 N Bodies in a Spherical Container

So far we have considered N bodies in an unbounded domain. In this section the boundary integral equations (18) 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 stays the same — the particle surfaces. 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 Section 4.1.3, except that the elements in the system matrix $\tilde{\boldsymbol{A}}$ and in the constant vector $\tilde{\boldsymbol{b}}$ have changed. Thus, the same iteration algorithm and its same implementation in the CSM model can be used. However, the routines for creating the system matrix $\tilde{\boldsymbol{A}}$ and the vector $\tilde{\boldsymbol{b}}$ 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. For the problem at hand, the relevant Green's function is [Jackson, 1975]

$$\bar{G}(\boldsymbol{x}, \boldsymbol{\xi}) = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} - \frac{a}{4\pi|\boldsymbol{\xi}| \left|\boldsymbol{x} - \frac{a^2}{|\boldsymbol{\xi}|^2}\boldsymbol{\xi}\right|} = \frac{1}{4\pi|\boldsymbol{x} - \boldsymbol{\xi}|} - \frac{a}{4\pi|\boldsymbol{\xi}|\,|\boldsymbol{x} - \boldsymbol{\xi}^*|} \ , \tag{22}$$

where $\boldsymbol{\xi}^* = \frac{a^2}{|\boldsymbol{\xi}|^2}\boldsymbol{\xi}$ is known as the inverse point. The first term is the usual free-space Green's function corresponding to a point charge at $\boldsymbol{\xi}$; the second term corresponds to an image charge at the inverse point that serves to cancel the first term on the sphere surface $|\boldsymbol{x}| = a$.

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 its 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:

$$K(\boldsymbol{x},\boldsymbol{\xi}) = 2\nabla_{\xi} G(\boldsymbol{x},\boldsymbol{\xi}) \cdot \hat{\boldsymbol{n}}(\boldsymbol{\xi}) \ . \tag{23}$$

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

$$\bar{G}(\boldsymbol{x},\boldsymbol{\xi}) = \frac{1}{4\pi}\left[\frac{1}{r} - \frac{\alpha}{r^*}\right] , \tag{24}$$

where

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

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

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

$$\frac{\partial}{\partial \xi_i}\frac{1}{r} = \frac{x_i-\xi_i}{r^3} , \tag{25}$$

$$\frac{\partial \alpha}{\partial \xi_i} = -\frac{a}{|\boldsymbol{\xi}|^3}\xi_i , \tag{26}$$

$$\frac{\partial \xi_j^*}{\partial \xi_i} = \frac{a^2}{|\boldsymbol{\xi}|^2}\left(\delta_{ij} - 2e_ie_j\right) \ . \tag{27}$$

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:

$$\bar{K}(\boldsymbol{x},\boldsymbol{\xi}) = \frac{1}{2\pi}\hat{n}_i\left\{\frac{x_i-\xi_i}{|\boldsymbol{x}-\boldsymbol{\xi}|^3} + \frac{|\boldsymbol{\xi}^*|^2}{|\boldsymbol{x}-\boldsymbol{\xi}^*|a^3}\left[-|\boldsymbol{\xi}^*|(\delta_{ij}-2e_ie_j)\frac{\boldsymbol{x}_j-\boldsymbol{\xi}_j^*}{|\boldsymbol{x}-\boldsymbol{\xi}^*|^2} + e_i\right]\right\} \ . \tag{28}$$

Note, that the first term of $\bar{K}(\boldsymbol{x},\boldsymbol{\xi})$ is the contribution of the original (free–space) double layer and its second term is all contributed from its image. Note also, that the boundary integral equations, such as (18), stay the same, except that the contribution of the image point charge has to be added and the kernel of the double layer integral has to be replaced. Thus, the boundary integral equation (18) turns into:

$$\begin{aligned} \varphi(\boldsymbol{\eta}) &= -\psi^{\infty}(\boldsymbol{\eta}) - \frac{1}{S_k}\oint_{S_k}\varphi(\boldsymbol{\xi})\,dS_{\xi} \\ &- \sum_{l=1}^{N}\left\{\frac{Q_l}{|\boldsymbol{\eta}-\boldsymbol{x}_l|} - \frac{aQ_l}{|\boldsymbol{x}_l|\left|\boldsymbol{\eta}-\frac{a^2}{|x_l|^2}\boldsymbol{x}_l\right|} + \oint_{S_l}\bar{K}(\boldsymbol{\eta},\boldsymbol{\xi})\varphi(\boldsymbol{\xi})\,dS_{\xi}\right\} \\ &\qquad \boldsymbol{\eta}\in S_k \qquad ; k = 1...N \ . \end{aligned} \tag{29}$$

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 $\tilde{\boldsymbol{A}}$ and the constant vector $\tilde{\boldsymbol{b}}$.

4.3 Algorithm

The linear algebraic equation system (20) derived in Section 4.1.3 has a large number of unknowns and a dense system matrix $\tilde{\boldsymbol{A}}$. For a problem of N bodies, which are discretized into M boundary elements each, the number of unknowns is MN. The number of elements in $\tilde{\boldsymbol{A}}$ is $(MN)^2$. 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 equation system (21) is solved by a parallel form of the *Jacobi Iteration* that can be applied because of the small spectral radius of the system matrix $\tilde{\boldsymbol{A}}$. The Jacobi Iteration can easily be 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 is 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.

The basic idea of the sequential Jacobi Iteration is to guess the solution $\boldsymbol{x}$ of the equation system (21) initially, then to insert this guess into the right hand side of Equation (21). Evaluating the right hand side gives a new approximation for the solution $\boldsymbol{x}$ that 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 on the ith boundary element on the kth surface φ_{ki} in iteration step t, then the initial guess is given by:

$$x_m(0) = b_m \ . \tag{30}$$

Inserting this initial guess into the right hand side of (21) we get a new more accurate guess $x_m(t+1)$.

$$x_m(t+1) = \sum_{n=1}^{NM} \tilde{A}_{mn} x_n(t) + b_m \qquad ; m = 1 \ldots NM \ . \tag{31}$$

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

To parallelize the Jacobi Iteration the solution vector $\boldsymbol{x}$ is split into N subvectors $\boldsymbol{y}_k$, where $\boldsymbol{y}_k$ represents the solution on the surface of one body k. 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 (31) as well as a subvector $\boldsymbol{d}_k$ of the constant vector $\tilde{\boldsymbol{b}}$, and is responsible for updating its set of sub solution vectors $\boldsymbol{y}_k$. Processor p has to evaluate the following formulas to update its sub vectors within one iteration step:

$$\boldsymbol{y}_k(t+1) = \begin{pmatrix} \boldsymbol{C}_{k1} & \boldsymbol{C}_{k2} & \cdots & \boldsymbol{C}_{kN} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{y}_1(t) \\ \boldsymbol{y}_2(t) \\ \vdots \\ \boldsymbol{y}_N(t) \end{pmatrix} + \begin{pmatrix} \boldsymbol{d}_k \end{pmatrix} \tag{32}$$

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

Here the matrix $\boldsymbol{C}_{kl}$ is a submatrix of $\tilde{\boldsymbol{A}}$ and describes the influence of body l onto body k. Note, that no other submatrix contains any information about this influence. The formulas (32) assume that processor p knows the solution on every body surface at iteration step t. This means, that all the solutions on the bodies that are not calculated on processor p have to be communicated to processor p at time t. This is a *synchronous* algorithm, since this communication synchronizes all processors at time t. The drawbacks of this synchronous approach are that it does not give us much freedom in modifying the iteration scheme, as described in the subsequent paragraphs, and introduces a large amount of communication and synchronization overhead, where the processors have to wait for each other and are in an idle state.

A different, better performing approach is an *asynchronous iteration* scheme where a sub solution vector on a distant body is communicated, as soon and only when it is needed. This guarantees that always the most recent solution vector is used, which accelerates the convergence of the solution as the Gauss-Seidel Iteration does in the sequential case [Bertsekas and Tsitsiklis, 1989]. 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 far apart bodies. Thus, we do not want to consider the influence from a far apart body 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 to 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 to a sub solution vector, if this sub solution vector 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* looking at the array of bodies. 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 accesses gobal 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. A straightforward approach for setting the elements s_{kl} is taking a feature which 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 criterions for the degree of influence between two bodies, e.g. as presented in [Fuentes and Kim, 1992], but our simple choice already speeds up the convergence time of the solution by orders of magnitude. The asynchronous iteration algorithm is illustrated in Figure 2.

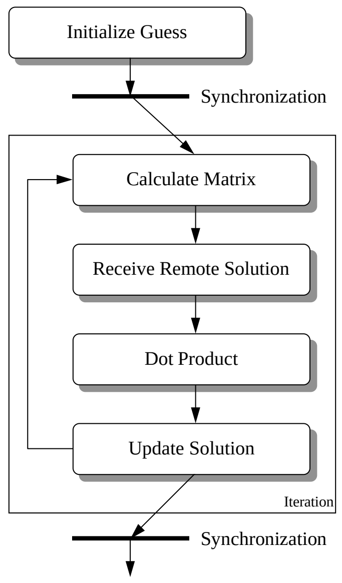

Figure 2: Asynchronous iteration algorithm.

4.4 Load Balancing

Load balancing has a large impact on our program's scaling performance. In order to optimize scaleup, the bodies are distributed to the processors in such a way that each processor has approximately the same workload. We use the CS matrix to estimate the amount of computation time needed to calculate the solution on one body, *viz.*, the sum of all entries in row k of the CS matrix $\boldsymbol{S}$ which corresponds to body k:

$$g_k = \sum_{l=1}^{N} s_{kl} \ . \tag{33}$$

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

$$g^p = \sum_{k=(p-1)N_P+1}^{pN_P} g_k \tag{34}$$

should have a minimal variance with respect to the processors. This problem wherein N nonnegative numbers, g_k, are partitioned into P blocks is called *off-line makespan scheduling* [Coffman and Lueker, 1991]. The sum of the numbers in one block p, g^p, has 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)* [Graham, 1969]. 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, the heuristic algorithm LPT is not optimal, but both its absolute and relative error tend to zero, if N is much larger than P [Coffman and Lueker, 1991].

4.5 Implementation in Cooperative Shared Memory

The parallel algorithm as described in Section 4.3 is implemented in Cooperative Shared Memory (CSM). There have already been two further implementations on the Thinking Machines Corporation's CM-5: one in ANSI C using the message-passing CMMD library and one in Split-C [Traenkle et al., 1993]. The CSM model fits our algorithm because of several reasons. First, the update of a sub solution vector $\boldsymbol{y}_l$ on processor p is done simply by accessing global shared memory. Second, the amount of read and write global shared data, the sub solution vectors $\boldsymbol{y}_k$ for instance, is small compared to the amount of read only local data, which is stored in the local memory spaces of the processors, such as the sub matrices $\boldsymbol{C}_{kl}$. And third, because of the distribution of the communication data in coherent memory blocks, one communication of one subvector $\boldsymbol{y}_l$ is done by one `check_in`, `check_out` pair.

The algorithm is easy to code in CSM because of the uniform address space of the global shared memory. Accessing data in global shared memory involves just one active processor, different from the message-passing implementation, where one processor is the requester and another processor is the server of the data (this requires the coding of request-and-send schemes, e.g. in the form of message loops).

CSM is a high level programming model and makes minor assumptions about the underlying hardware architecture. This makes it easy to port the code to different parallel machines with little or without any modifications.

5 Results and Discussion

5.1 *Dir*$_1$*SW*: An Implementation of Cooperative Shared Memory

The results in this paper assume that cooperative shared memory is implemented on a shared-memory computer using a cache-coherence protocol called *Dir*$_1$*SW*[‡] [Hill et al., 1993]. Recall that

[‡]The name *Dir*$_1$*SW* is derived by extending the directory protocol taxonomy as in [Agarwal, et al., 1988]. They use *Dir*$_i$B and *Dir*$_i$*NB* to stand for directories with i pointers that do or do not use broadcast. The *SW* in *Dir*$_1$*SW* stands for our *S*oft*W*are trap handlers.

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 1: Dir_1SW assumptions.

cache coherence prevents a processor from reading a cached copy of data when another processor has updated the original memory location. Dir_1SW uses hardware similar to that of a message-passing computer (c.f., Figure 1) plus it logically associates a small amount of memory–called a *directory entry*–with each cache-block-sized piece of the memory. Dir_1SW uses the directory entry to record that the block is idle (no cached copies), one writable copy exists, or 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.

Our numerical results also use the specific implementation assumptions listed in Table 1.

5.2 The Wisconsin Wind Tunnel

One challenge for performing experiments on the Dir_1SW computer is that it does not (yet) exist. 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 [Reinhardt et al., 1993]. WWT runs parallel shared-memory programs on a parallel message-passing computer (a Thinking Machines CM-5) [Thinking Machines Corp., 1991] 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 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 [Reinhardt et al., 1993]. 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. We guard against important errors by validating against existing systems, but we can never be sure our predictions are error-free at new design points.

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 cache size. Third, we get repeatable performance predictions and access internal hardware state to determine where time was wasted. Fourth, we can build virtual prototypes for other computers and compare results in controlled experiments.

The net result is that the Wisconsin Wind Tunnel allows us to run on future parallel supercomputers to influence how they are built! WWT allows us to cull the parallel supercomputer design space in a manner similar to how aeronautical engineers use conventional wind tunnels to design airplanes. This is why WWT is so named, even when it does not blow air.

5.3 Convergence Time

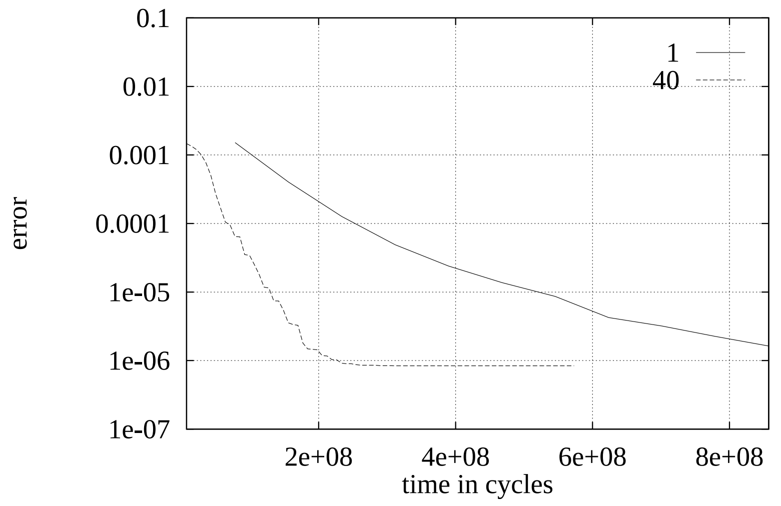

Figure 3: Relative error versus execution time in clock cycles.

This section discusses the execution time of the algorithm in Section 4.3 with respect to different communication scheme (CS) matrices. In Figure 3 two program runs with two different

Number of processors	64
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 $\max_{k,l} s_{kl}$	1, 40

Table 2: Input parameters for convergence time.

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 3: Input parameters for speedup.

CS matrices $\boldsymbol{S}$ are presented. The relative error of the solution vector with respect to a solution obtained in a previous run at a high iteration step versus the execution time in clock cycles is presented.[†] The input parameters common to both runs are presented in Table 2.

Curve "1" represents a run where all sub solution vectors are updated on every processor in every iteration step, which corresponds to an algorithm using a synchronous iteration scheme, that is, all entries in $\boldsymbol{S}$ are set to one. Curve "40" is a run where the element s_{kl} corresponding to the most distant bodies k and l is set to 40 and other elements are scaled w.r.t. to their corresponding body distances. Since this run communicates the sub solution vectors between the different bodies less frequently and less recent data is available on the processors, the number of iteration steps needed for convergence increases. However, the execution time decreases significantly because of less computation and less communication. Both runs store $\frac{240}{256}$ of the system matrix in memory permanently before the execution of the iteration, the remaining $\frac{16}{256}$ is recalculated during the iteration when needed, which is the main part of the execution time. Submatrices describing far field interactions are least likely to be stored permanently, because they are recalculated very infrequently in case of run "40". Run "40" converges to the same solution for the double layer density as in run "1", but at a much faster rate (in time).

5.4 Speedup

An important criterion for the performance and quality of a parallel program is its *speedup*. The speedup measures the scaling of the program under the contraint of a constant problem size. The program is executed on different numbers of processors but with the same input set. The speedup is usually defined as

$$S_P(n) = \frac{T^*(n)}{T_P(n)} \ , \tag{35}$$

where n is the size of the given input set, P the numbers of processors, T_P the execution time on P processors and $T^*(n)$ the execution time of the optimal sequential implementation of the algorithm. However, $T^*(n)$ is unknown for our algorithm and the problem with 256 bodies is too

[†]A 100 MHz microprocessor, for example, takes one second to execute 10^8 clock cycles.

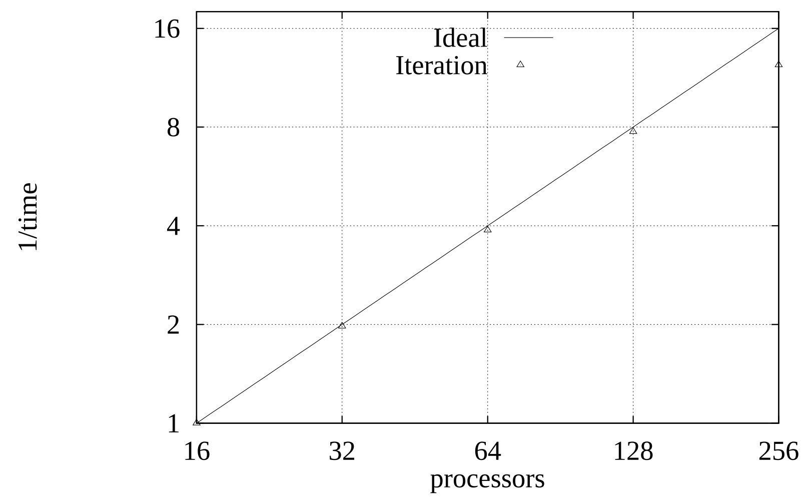

Figure 4: Speedup w.r.t. 16 processors.

large for the memory and speed of one processor. So we use a modified speedup based on the execution time T_{16} on 16 processors:

$$\bar{S}_P(n) = \frac{T_{16}(n)}{T_P(n)} \ . \tag{36}$$

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 input set described in Table 3 the iteration algorithm has been executed on 16, 32, 64, 128, and 256 processors. The resulting speedup is shown in Figure 4 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 (the ideal speedup is 8). 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 4), 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.

The use of 16–processor data as the base point for speedup computations brings up an important point concerning performance evaluations spanning several orders of magnitude in numbers of processors. The classical definition of speedup requires results from a one–processor run, but it is impractical to evaluate large simulations with a single–processor run; most notably because even if total memory used is fixed, the memory hierarchy intrinsic to high performance computing complicates the performance analysis. The alternative of running a smaller problem with one processor sidesteps the issue of scalability of a given large scale simulation. The ultimate resolution of this issue lies in a more sophisticated definition of speedup and scaleup. In the present paper, the 16–processor base point serves the purpose of illustrating the scalability of the present work to large numbers of processors.

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 4: 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*.

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 5: Input parameters for time-constraint scaleup.

5.5 Time-constraint Scaleup

Another measure for the performance of a parallel program is its scaleup under the constraint of constant execution time. The execution time is estimated *a priori* for different input sets and for different numbers of processors. As in the case of the speedup measurement, the program is executed with different numbers of processors, but now the input sets are chosen in such a way that the execution time stays constant. The actually obtained execution time is a measure for the scaling of the program. In the ideal case, this execution time scaled w.r.t. the number of processors in the smallest used partition is equal to 1. Since the calculation time of the system matrix is of $O((NM)^2)$ and dominates the iteration time, the overall computation time of the iteration algorithm, assuming that parts of the system matrix are recalculated within the iteration, is also of $O((NM)^2)$.

According to this order the input sets for the runs on the different partition sizes are chosen as in Table 5. The rest of the parameters, for instance the number of boundary elements, are as shown in Table 3. Because of the limitations for the partition sizes there are two different sets of runs both presented in Figure 5, which show the almost ideal scaleup of the iteration algorithm. However, the scaleup is much worse for the run on 256 processors. This is due to the effect already discussed in Section 5.4, where the degree of freedom in distributing the bodies to get a good load balance is zero.

5.6 Many Bodies in a Spherical Container

Using the solution for the double layer density returned by the algorithm discussed so far, and Equation (15) the potential is calculated on a planar cross section through the spherical container. We considered an array of randomly placed 64 equally sized and equally charged spheres within a

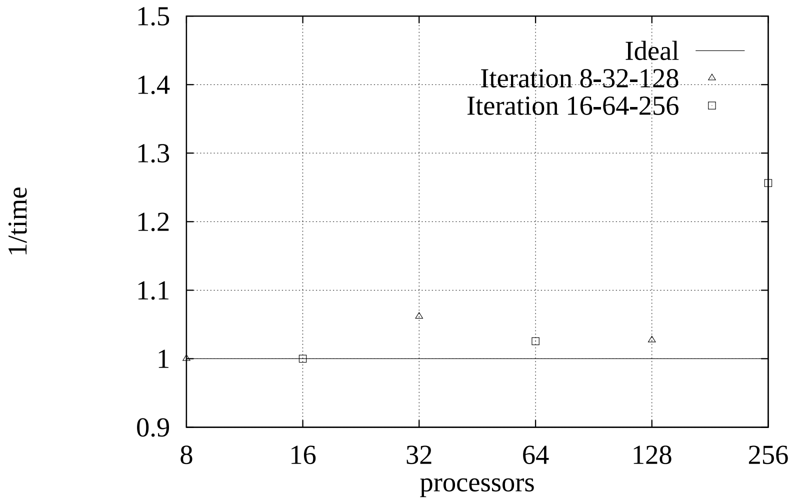

Figure 5: Time-constraint scaleup.

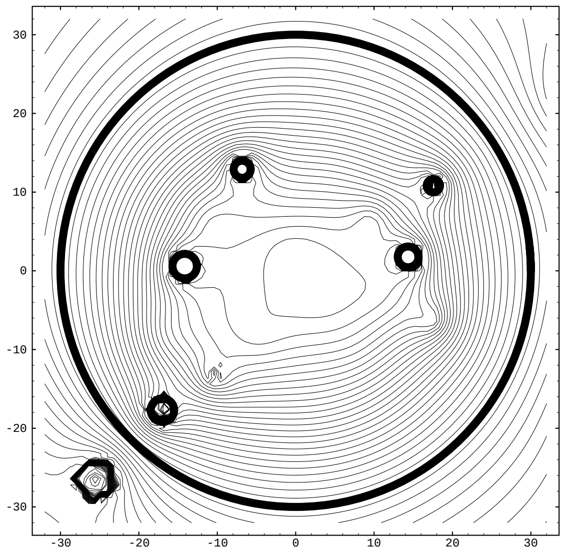

Figure 6: 64 bodies in a container.

spherical container, whose surface is an equipotential with potential 0, and calculated the potential lines on this crossection as shown in Figure 6. The thick lines are the boundaries of the container and of the spheres, the thinner lines are the equipotentials. In the future, routine access to this level of computational modeling allow us to bridge the gap between idealized microstructure theories (e.g. periodic lattice models) and macroscopic semi-empirical theories (e.g. cell models).

6 Conclusions

In several respects, the situation we face today for coding of high performance parallel algorithms is analogous to the difficulties faced by computational scientists in the early days of the electronic digital computer. To obtain optimal performance, the programs had to be written in machine code and were not portable from one machine to another. This observation is particularly relevant for computationally oriented chemical engineers who, while interested in harnessing the power of massively parallel computers, are not willing to devote inordinate amounts of time to rewrite their codes for every successive generation of parallel machines. The need for a generally accepted parallel programming model, to pave the way for high–performance high–level parallel programming language(s) is quite clear. Such a programming model would play a role analogous to that played by Fortran in the development of the digital computer, in making computational power accessible to a larger community of scientists and engineers.

Our work contributes to this goal. A parallel programming model (CSM) has been proposed, and its performance should be tested on a number of "typical" scientific and engineering applications. The CDLBIEM algorithm has a record of good performance on small shared-memory (Sequent Symmetry) and message-passing computers (Intel iPSC/860, CM5) that provides a basis for comparison. Our finding is that coding is much simpler on CSM than with the message-passing model (in fact, quite similar to our experience on the Sequent Symmetry), and yet performance (computational times and speed ups) is comparable, even when scaled up to 256 processors. This result may be of great interest to designers of future machines.

Because of space limitations, we have said very little about the physical applications of these computations. We merely note in passing that this class of methods is useful in bridging the gaps in length and time scales between macroscopic, empirical theories (such as cell models) and microscopic theories that incorporate microstructure information on a more rigorous manner. Readers interested in an example of such linkages are directed to a companion article [Traenkle and Kim, 1994].

Since CSM machines do not exist (yet), our results were obtained on the Wisconsin Wind Tunnel. An important by-product of this direction taken is that we were also able to examine performance as a function of machine parameters such as cache size and network bandwidth and latency. In essence, we were able to vary simultaneously the algorithm and architecture, to look for optimal combinations. Given the enormous cost of building actual hardware, we believe that this concept will become an important area for further research in the parallel computing community.

7 Acknowledgements

We would like to thank members of the Wisconsin Wind Tunnel Project—Babak Falsafi, Alvin Lebeck, Jim Larus, Shubu Mukherjee, Steve Reinhardt, and David Wood—for many thought-provoking discussions and assistance using WWT. We also thank Matt Frank for helpful discussions on communication scheduling.

The first author also thanks the DAAD for financial support. This work is supported in part by NSF PYI Awards CCR-9157366 and MIPS-8957278, NSF Grants CCR-9101035 and CTS-9218668, ONR Grant N00014-92-J-1564, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories and Digital Equipment Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

Nomenclature

Roman letters

$\boldsymbol{A}$	system matrix
$\tilde{\boldsymbol{A}}$	system matrix
$\boldsymbol{C}_{kl}$	sub system matrix
$\boldsymbol{E}^\infty$	electric field at infinity
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	domain of the fluid or vacuum
a	sphere radius
$\boldsymbol{b}$	constant system vector
$\boldsymbol{d}_k$	constant system subvector
$\boldsymbol{n}$	surface normal pointing into the body
$\hat{\boldsymbol{n}}$	surface normal pointing into the fluid or void
s_{kl}	element of CS matrix
$\boldsymbol{x}$	position vector
$\boldsymbol{x}$	solution vector
$\boldsymbol{y}_k$	sub solution vector on body k
$\boldsymbol{x}_\alpha$	position of singularity in body α

Greek letters

δ	Dirac's delta function
δ_{ij}	Kronecker delta
$\boldsymbol{\eta}$	position vector on body surface
$\boldsymbol{\xi}$	position vector as integration variable
φ	double layer density
$\varphi^{(\alpha)}$	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. $\boldsymbol{x}$
∇_{ξ}	gradient w.r.t. $\boldsymbol{\xi}$
$\nabla\cdot$	divergence

Subscripts

i, j	boundary element label
k, l	body label
α	body label

Superscripts

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

References

A. Agarwal, R. Simoni, M. Horowitz and J. Hennessy, *An Evaluation of Directory Schemes for Cache Coherence*, ISCA88, 280–289 (1988).

N. Amann and S. Kim, *Parallel Computational Microhydrodynamics: Load Balancing Strategies*, Engineering Analysis with Boundary Elements, **11**, 269–276 (1993).

D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation - Numerical Methods*, Prentice Hall, (1989).

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).

E.G. Coffman and G.S. Lueker. *Probalistic Analysis of Packing and Partitioning Algorithms.* Wiley, New York, (1991).

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), 1–12 (May 1993).

M.J. Flynn. *Very High–speed Computers.* Proc. IEEE, **54**, 1901–1909 (1966).

A. Friedman. *Foundations of Modern Analysis.* Dover, New York (1982).

Y.O. Fuentes and S. Kim. *Parallel Computational Microhydrodynamics: Communication Scheduling Strategies.* A.I.Ch.E. Journal, **38**, 1059–1078 (1992).

G.L. Graham. *Bounds on Multiprocessing Timing Anomalies.* SIAM Journal on Applied Mathematics, **17**, 263–269 (1969).

S.R. Graubard (ed.). *A New Era in Computation.* DAEDALUS J. Amer. Acad. Arts and Sciences. Winter, (1992).

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, **11**, 300–318 (1993).

J.D. Jackson. *Classical Electrodynamics.* Wiley, New York, (1975).

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).

Kendall Square Research. *Kendall Square Research Technical Summary.* (1992).

S. Kim and S.J. Karrila. *Microhydrodynamics: Principles and Selected Applications.* Butterworth–Heinemann, Boston, (1991).

D.J. Klingenberg, F. van Swol, and C.F. Zukoski. *Dynamic Simulation of Electrorheological Suspensions.* J. Chem. Phys., **91**, 7888–7895 (1989).

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).

C. Lin and L. Snyder. *A Comparison of Programming Models for Shared Memory Multiprocessors.* In Proceedings of the 1990 International Conference on Parallel Processing (Vol. II Software), pages II–163–170, (August 1990).

N. Phan-Thien and S. Kim. *Microstructures in Elastic Media.* Oxford University Press, New York, (1994).

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).

R. Rettberg, and R. Thomas. *Contention is no Obstacle to Shared Memory Multiprocessing.* Communications of the ACM, **29**, 1202–1212 (1986).

Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary.* (1991).

F. Traenkle, M.I. Frank, M.K. Vernon, and S. Kim. *Solving Microstructure Electrostatics with MIMD Parallel Supercomputers and Split-C.* In J. Non-Newtonian Fluid Mech., (1994) in press.

F. Traenkle, B.E. Saunders and S. Kim. *Effective Conductivities of Composites with Spherical Inclusions: Periodic and Microstructured Cell Models.* In A.I.Ch.E. J., (1994) submitted.