Parallel Computational Microhydrodynamics: Communication Scheduling Strategies

Yuris O. Fuentes
Dept. of Chemical Engineering, University of Colorado, Boulder, CO 80309

Sangtae Kim
Dept. of Chemical Engineering, University of Wisconsin, Madison, WI 53706

The behavior of multiparticle systems in a viscous fluid, as governed by the Stokes equations, is computed by the coordinated use of multiple processors on a distributed memory parallel computer. The completed double-layer boundary integral equation method (CDL-BIEM) is used to convert the hydrodynamic mobility problems to a fixed-point problem, amenable to either synchronous or asynchronous iterative solution schemes. Parallel computational strategies, based on assigning particles to processors, are considered, and rules are derived to specify how often processors should exchange information. A spectral communication scheduling strategy, based on the spectral radius in pair-interaction problems, converges with fewer global iterations and effectively reduces the level of interprocessor communications, suggesting algorithm scalability to massively-parallel computers with hierarchical access to distributed memories. Stochastic schedules, which specify the probability that the information exchange occurs at every iteration, were also considered. For the same test problems, these strategies performed better than the point Jacobi iterations, but not as well as their deterministic counterparts. Scheduling strategies are extrapolated to larger problems, based on projections of memory and performance capabilities of the next generation of high-performance parallel supercomputers.

Introduction

Computer simulation of microstructure evolution promises to play an increasingly important role in our efforts to create new materials and more efficient materials processing techniques. But the three-dimensional flows involving such complex fluid-solid boundaries do and continue to present formidable challenges to the high-performance computers of today and tomorrow. If the discipline of "computational simulations" is to play a meaningful role in the engineering triad of theory-experiment-simulation, we must first develop faster computers and faster algorithms, since many of the key simulation benchmarks are beyond the reach of current computers by at least 6 orders of magnitude (as measured by computational time). Moreover, fundamental physical constraints such as the speed of light and VLSI linewidths make it unlikely that we will meet this goal with conventional computer architectures. The most promising direction is parallel computing: the coordinated use of thousands (and perhaps millions) of powerful "supercomputers-on-a-chip," but only if the simulation algorithms can be broken down to many independent tasks that are amenable to concurrent computation. In this article, we consider one of the most fundamental equations of chemical engineering science, the Stokes equations of viscous hydrodynamics, and map their solution algorithms onto a scalable parallel architecture so that in the near future, quite complex simulations of microstructure evolution in the dispersed multiphase system (suspensions, composite materials, and so on) may be attempted.

Particle motion in a viscous fluid

Determining the motion of small particles moving in reaction to applied external forces and interparticle forces is widely

Correspondence concerning this article should be addressed to S. Kim.

recognized as a key problem in our efforts to link multiphase microstructure and bulk material phenomena, with applications ranging from predicting bulk material properties such as suspension viscosity to improved materials processing via better control of microstructure. For an assembly of particles dispersed in a vacuum, the governing equations follow directly from Newton's laws of motion; given all forces acting on the particles, the particle trajectories in principle can be determined by integration of the acceleration field. Molecular dynamic simulations originate from this simple idea. On the other hand, when the multiphase system comprises particles dispersed in a viscous fluid, the situation becomes far more complicated, since particle and fluid motions are now coupled. Given all forces acting on the particles, we must also determine the resulting motion of the fluid as governed by the appropriate partial differential equation that follows from the conservation laws for mass, momentum and energy. This article describes recent progress in tackling this fundamental problem for model systems, in which the key approximations are not too unrealistic and appear quite reasonable as applied to commonly encountered suspension microstructures.

We consider a suspension microstructure consisting of small rigid particles dispersed in an incompressible Newtonian fluid of viscosity μ. Furthermore, the system is assumed to be isothermal, with energy dissipated by viscous effects removed rapidly to the ambient with a negligible rise in the temperature. Finally, for the range of particle sizes of greatest interest (say 0.01 μm-100 μm) and common viscous fluids, fluid inertia is negligible compared to viscous forces for the range of particle velocities that are of interest (the particle-based Reynolds number is essentially zero). Under these circumstances, the governing equations from momentum and mass conservation, which describe the fluid flow, reduce to the Stokes equations:

$$-\nabla p+\mu\nabla^2 v=0, \quad \nabla\cdot v=0. \tag{1}$$

The coupling between particle motion, viz., the rigid body motion of particle α, $U_\alpha+\omega_\alpha\times(x-x_\alpha)$, and fluid motion comes in via the no-slip boundary conditions at the particle surface S_α:

$$v=U_\alpha+\omega_\alpha\times(x-x_\alpha) \quad x\in S_\alpha. \tag{2}$$

In the original problem statement, we are actually given the net external forces and torques acting on the particle, and the particle translational velocity U_α and rotational velocity ω_α used in the above boundary conditions are thus unknowns. However, the solution procedure used here will obtain the fluid velocity field and simultaneously determine the rigid body motions associated with the given set of external forces and torques acting on each particle, that is, solve the mobility problem. (For particles in the submicron range, Brownian motion, which results from the cumulative effect of many collisions with solvent molecules, cannot be neglected. Even for this case, however, Eqs. 1 and 2 constitute an important step, since mobilities obtained therein appear as coefficients in the governing stochastic differential equation.)

Our goal is to elucidate the evolution of suspension microstructure by providing new computational algorithms for solving particulate Stokes flow problems, if at all possible, from first principles. Examples include situations such as N (where N can be quite large) particles sedimenting in a container and suspensions of infinite extent modeled as periodic media, with large unit cells containing many particles. We place no restrictions on the shape of rigid particles or the container boundary.

Since we are dealing with a three-dimensional problem with complex geometry, inevitably we face a large-scale computational task. These computations are actually beyond the capabilities of present computers, but we will make the case in this work that new solution methods map quite naturally onto high-performance parallel computers of the near future.

Emerging high-performance parallel supercomputers

We are approaching a major turning point in the evolution of computers, with broad implication for the role of computers in science, engineering, and technology. New opportunities exist that are perhaps comparable in scope to those that emerged with digital electronic computers three decades ago. The performance of a single fast superprocessor is ultimately bound by fundamental physical constraints, such as the speed of light. Given that grand challenges in science, engineering, and technology demand computational powers that greatly exceed current capabilities (perhaps by a factor of 10^6), it is apparent that a new paradigm is needed. Figure 1 shows the evolution of supercomputer and "microprocessor" floating point capabilities during the past decade (see Dongarra, 1990; projections in Deng et al., 1992). The ordinate depicts on a logarithmic scale, performance in MFlops (million floating point operations per second). On the scale used in this figure, the respectable advances in conventional supercomputer capabilities dim in comparison to the dramatic improvements in the floating point performance on a square-inch chip of silicon. Indeed, the peak speed of 60 MFlops of the state-of-the-art Intel i860 is within an order of magnitude of the 100MFlop benchmark—the traditional unit measure of supercomputing performance. With shrinking semiconductor dimensions, it is quite likely that in the near future, the square inch of silicon will house four and then 16 such processors. Figure 1 extrapolates the upward slope of the semiconductor processor curve during this decade.

With the rapid development of supercomputing at the chip

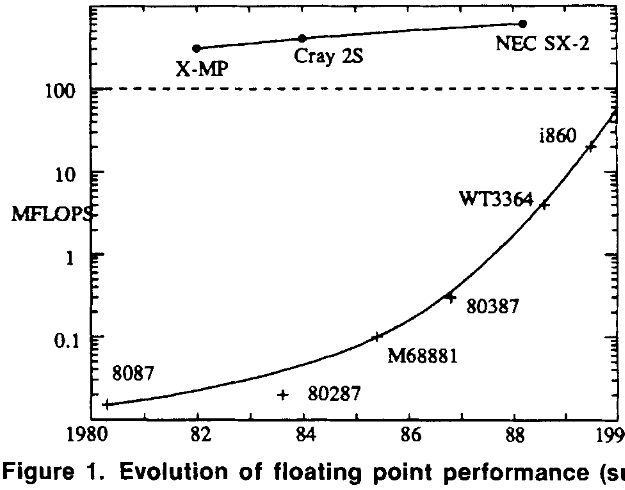

Figure 1. Evolution of floating point performance (supercomputer and micro) during the 80's: 1000 × 1000 LINPACK, from Dongarra (1990).

level, the most promising approach lies with computer architectures that are essentially a network of many tightly integrated "supercomputers on a chip, or what is commonly called *a high-performance parallel computer*. One can envision in the near future a computing environment consisting of over 16,000 processors with an aggregate performance exceeding the TFlop (one trillion floating point operations per second) level, but only for highly parallel algorithms. The challenge then is twofold: map currently popular solution methodologies to parallel algorithms; develop new solution methods that naturally lead to parallel algorithms. This work may be summarized in one sentence: *We continue to develop the completed double-layer boundary integral equation method (CDL-BIEM) for particulate Stokes flow problems and introduce a number of ideas for scaling up CDL-BIEM onto a many-processor environment, with particular emphasis on overcoming interprocessor communication bottlenecks and memory limitations.* We have reserved more space to describe these new ideas, especially since the CDL-BIEM algorithm is described in great detail elsewhere. Only key CDL-BIEM equations will be stated here. Readers who are interested in the evolution of CDL-BIEM are directed to the original articles by Karrila and Kim (1989) and Karrila et al. (1989), or the more complete exposition in the *Microhydrodynamics: Principles and Selected Applications* by Kim and Karrila (1991).

This article starts with a brief summary of the basic concepts of parallel computing, with the goal of setting the stage for illustrating the link between scalable algorithms and scalable architectures. We discuss the terminology and the main issues that do not come up in traditional sequential computing (such as communications and synchronization). Next, equations used in the boundary integral formulation for many-body interactions in a viscous fluid are discussed, as well as asynchronous algorithms for iterative solution of linear systems. Having discussed the underlying computational and theoretical aspects, we present the design and implementation of communication scheduling strategies for the solution of multiparticle system in Stokes flow on distributed memory parallel computers with specific results for the Intel iPSC/860. The last section includes the main conclusions of our work and ideas on future directions for research on massively parallel computers.

Parallel Computing

In this section several concepts related to parallel and distributed computations will be discussed. A general overview will be given; the interested reader may refer to a number of excellent references on this subject (Bertsekas and Tsitsiklis, 1989; Brawer, 1989; Stone, 1987, for example) for a comprehensive treatment covering algorithms, architectures and programming.

There are several issues related to parallelization that do not come up in the serial (or sequential) context. The first issue is *task allocation*, that is, the process of splitting the total workload in smaller tasks assigned to different processors and properly sequencing these tasks when some of them are interdependent and cannot be executed simultaneously. A second issue is *communication* of intermediate results between processors—its efficiency and impact on performance. A third issue is the *synchronization* of computations of different processors. The degree of synchronicity can vary, from *synchronous* methods, in which processors must wait at predetermined points for the completion of certain tasks or for the arrival of certain data, to *completely asynchronous* methods, in which there are no requirements of synchronism and corresponding implications of the methods' validity must be assessed. Other issues relate to the development of appropriate performance measures for parallel applications and the effect of the system's architecture on these performance measures.

Parallel and distributed architectures

We will first discuss the distinction between *parallel* and *distributed* computer systems. Essential differences between parallel and distributed systems are related to the connectivity between the processors. Parallel computing systems consist of several processors that are located within a small distance of each other, generally within the same cabinet. They have been designed to execute jointly a computational task, thus communication between the processors is reliable and predictable. In distributed systems, the processors may be far apart and interprocessor communication can become a problem. Communication delays may be unpredictable and the communications themselves may be unreliable. Distributed systems are loosely coupled and there is very little, if any, central coordination. Each processor performs its own private activities, while at the same time cooperating with other processors in the context of some global computational task. Furthermore, the topology of a distributed system may undergo changes, while the system is operating, for example, due to addition or removal of processors, or failures of communication links. Therefore, while the architecture of a parallel system is under the control of the system designer, the structure of some distributed system is dictated by exogenous considerations. However, there are several algorithmic issues that are similar, and at that level there is no clear dividing line between parallel and distributed systems and both terms can be used interchangeably.

There are several parameters or characteristics that can be used to classify or describe a parallel computer.

Type and Number of Processors. *Coarse-grained parallelism* refers to systems with a small number of fairly powerful processors, say of the order of 16. The processors are loosely coupled, in the sense that each processor may be performing a different task at any given time. At the other end, there are parallel systems with thousands of processors. Such systems are said to be *massively parallel.*

Presence or Absence of a Global Control Mechanism. Parallel computers, like traditional serial computers, always have some central locus of control. The difference is to what extent is the operation of individual processors controlled. At one extreme, a global control mechanism is used to load the program and data to the processors, and each processor works on its own thereafter. The degree of control, however, can get to the extreme where each processor is instructed what to do at every step. Intermediate situations are also conceivable. Another classification along these lines distinguishes between SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data) parallel computers. This refers to the ability of processors to execute different instructions at a given point in time.

Synchronous vs. Asynchronous Operation. The distinction here refers to the presence or absence of a common global clock used to synchronize the operation of different processors. For a system operating synchronously, the behavior of processors is much easier to control and algorithm design is considerably simplified, but at that same time, it may require an undesirable overhead.

Interprocessor Communications. An important aspect of parallel computers is the mechanism by which processors exchange information. Generally speaking, there are two extreme alternatives known as *shared memory* and *message-passing* architectures, and a variety of hybrid designs lying in-between. The first design uses a global shared memory that can be accessed by all processors. Processors communicate by writing into and reading from the global memory. This solves the interprocessor communication problem, but introduces the problem of simultaneous accessing different locations in memory by several processors. There are ways to handle this problem (such as *switching systems*), but they lead to longer memory access times, which increase with the number of processors. At the other extreme, there is no shared memory, but rather each processor has its own *local memory*. Processors communicate through an interconnection network consisting of direct communication links joining certain pairs of processors. It would be best if all processors were directly connected to each other, but this often is not feasible: either there is an excessive number of links leading to increased costs or processors communicate through a shared bus leading to excessive delays due to bus contention when the number of processors is large.

Performance issues

We now describe several concepts that are sometimes useful in comparing serial and parallel algorithms. Consider a computational problem parameterized by a variable n representing the problem size. Suppose we have a parallel algorithm that uses p processors (p could depend on n) and that terminates in time $T_p(n)$. Let $T^*(n)$ be the optimal serial time to solve the same problem, that is, the time required by the best possible serial (uniprocessor) algorithm for the problem. The ratio

$$S_p(n) \doteq \frac{T^*(n)}{T_p(n)} \tag{3}$$

is called the *speedup* of the algorithm. It describes the speed advantage of the parallel algorithm compared to the best serial algorithm. The ratio

$$E_p(n) \doteq \frac{S_p(n)}{p} = \frac{T^*(n)}{pT_p(n)} \tag{4}$$

is called the *efficiency* of the algorithm, and it essentially measures the fraction of time during which a typical processor is usefully employed. Ideally, $S_p(n) = p$ and $E_p(n) = 1$, in which case, the availability of p processors allows us to speed up the computation by a factor of p. This ideal situation is practically unattainable. A more realistic goal is to aim at an efficiency that stays bounded away from zero, as n and p increase.

There is practical difficulty with the definitions given above because the optimal serial time $T^*(n)$ in general is unknown. For this reason, $T^*(n)$ is sometimes defined differently. Some alternatives are:

(*a*) Let $T^*(n)$ be the time required by the best existing serial algorithm.

(*b*) Let $T^*(n)$ be the time required by a benchmark serial algorithm.

(*c*) Let $T^*(n)$ be the time required by a single processor to execute the particular parallel algorithm being analyzed $T_1(n)$.

In evaluating the performance of our algorithms, we will be using alternative c. With this choice of $T^*(n)$, we let a single processor simulate the operation of p parallel processors, and therefore the efficiency relates to how well a particular algorithm has been parallelized (with respect to communication and synchronization). Note, however, that it provides no information on the absolute merits of the algorithm [in contrast with the other definitions of $T^*(n)$].

A fundamental issue is whether the maximum attainable speedup

$$T_1(n)/T_\infty(n)$$

can be made arbitrarily large, as n is increased. The main difficulty is that in general programs have sections that are easily parallelizable, but also have some sections that are inherently sequential. When a large number of processors are available, the parallelizable sections are executed quickly, but the sequential sections lead to bottlenecks. This observation is known as *Amdahl's law* 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 computation, the attainable speedup is bounded by:

$$S_p(n) \leq \frac{1}{f + (1-f)/p} \leq \frac{1}{f}, \quad \forall p. \tag{5}$$

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

Communications issues

In many parallel and distributed algorithms and systems, the time spent for interprocessor communication is an important fraction of the total time needed to solve the problem. In such cases, we say that the algorithm experiences substantial communication penalty or communication delays. We can think of the communication penalty as the ratio:

$$CP \doteq \frac{T_{\text{total}}}{T_{\text{comp}}}, \tag{6}$$

where T_{total} is the time required by the algorithm to solve a given problem, and T_{comp} is the corresponding time that can be attributed just to computation, that is, the time that would be required if all the communications were instantaneous.

Some of the most important factors that affect the communication delays are the following:

1. The algorithm used to control the communications network, mainly error control, routing, and flow control

2. The communication network topology, that is, the number, nature and location of the communication links

3. The structure of the problem solved and the design of the algorithm to match this structure, including the degree of synchronization required by the algorithm.

At the user level, nothing can be done about items 1 and 2. Clearly, the design of an efficient solution algorithm to match the available architecture (including synchronization aspects, which will be discussed later) becomes critical in minimizing the communication penalty. It is important, however, to understand the effect of the network topology on communications, and therefore on the speedup and efficiency of a given algorithm.

Network topologies

A communication network of processors can be represented as a graph $G=(N, A)$, also referred to as a *topology*. The nodes (N) of the graph correspond to the processors, and the presence of an (undirected) arc (i, j) (A) indicates that there is a direct communication link between processor i and processor j.

Topologies are usually evaluated in terms of their suitability for some standard communication tasks. The following are some typical criteria:

- The *diameter* of the network is the maximum distance between any pair of nodes. The distance between a pair of nodes is the minimum number of links that have to be traversed to go from one node to the other.
- The *connectivity* of the network provides a measure of the number of independent paths connecting a pair of nodes. We can talk here about node or arc connectivity, which is the minimum number of nodes (or arcs, respectively) that must be deleted before the network becomes disconnected.
- The *flexibility* provided in running efficiently a broad variety of algorithms. A topology is flexible if many other topologies can be mapped into it.
- The *communication delay* required for some standard tasks related to information transfer between nodes that are important in many algorithms (such as inner product computation, matrix-vector multiplication).

We will briefly describe a number of specific topologies.

Complete Graph. Here there is a direct link between every pair of processors. Such a network can be implemented by means of a bus that is shared by all processors or by means of some type of crossbar switch. This is an ideal network in terms of flexibility. Unfortunately, when the number of processors is very large, a crossbar switch becomes very costly, and a bus involves large queuing delays.

Linear Processor Array. Here there are p processors/nodes, and there is a link between every pair of successive processors. The diameter and connectivity properties of this network are the worst possible. Furthermore, since this topology can be mapped into most networks of interest, the communication penalty for a given algorithm using a linear array can be no better than the corresponding penalty using most other networks.

Ring. This is a simple and common network with the property in which there are two paths between any pair of processors. All the basic communication problems can be solved on a ring in a time that lies between the corresponding time on a linear array with the same number of nodes and one-half that time.

Tree. A tree network with p processors provides communication between every pair of processors with a minimal number of links. One disadvantage of a tree is its low connectivity; furthermore, depending on the particular tree used, its diameter can be as large as p-1 (that of a linear array, which happens to be a special case of a tree). A star network has minimal diameter among tree topologies; however, the central node of the star has to handle all the network traffic and can become a bottleneck.

Mesh. In a d-dimensional mesh, processors are arranged along the points of a d-dimensional space that have integer coordinates, and there is a direct communication link between nearest neighbors. The diameter of a mesh-connected network can be much smaller than that of a ring and much larger than the diameter of a binary balanced tree with the same number of processors.

Hypercube. Consider the set of all points in a d-dimensional binary space. These points may be thought of as the corners of a d-dimensional cube. Let these points correspond to processors, and consider a communication link for every two points differing in a single coordinate. The resulting network is called a *hypercube* or d-cube, as shown in Figure 2. A hypercube is a flexible architecture—a ring and a mesh can be easily mapped into it. Not all trees, however, can be mapped to a d-cube. Due to the high connectivity, simultaneous communication between two nodes of a hypercube along several paths can be done efficiently.

Concurrency

We will conclude this section by discussing concurrency and its effect on the communication penalty. The term *concurrency* is used as a broad measure of the number of processors that are, in some aggregate sense, simultaneously active in carrying out the computations of a given parallel algorithm. 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 be relatively uniform across processors; otherwise, some processors will be idle waiting for others to finish their subtasks. This aspect of parallel computing is known as *load balancing*. In many cases, the number of packet exchanges

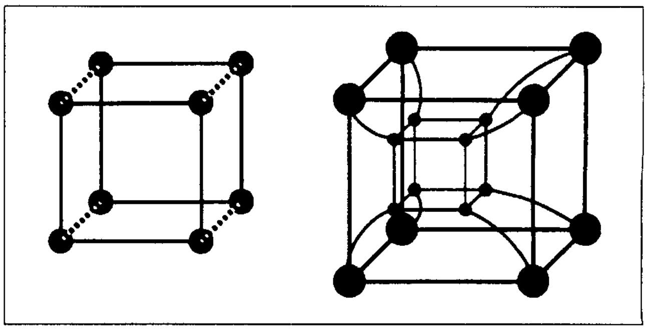

Figure 2. Hypercubes of dimension 3 and 4.

The dotted lines shown indicate how to go from dimension 2 to 3.

used to coordinate the parallel subtasks increases as the number of subtasks increases. Therefore, as the concurrency of an algorithm increases, the communication penalty for the algorithm also increases. Thus, as we attempt to decrease the solution time of a given problem 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 in *phases*. During each phase, every processor must perform a number of computations that depend on the results of computations of other processors in previous phases. The timing of the computations at any one processor, however, can be independent of that of other processors. That is, within a phase, each processor does not interact with other processors as far as that given algorithm is concerned. All interaction takes place at the end of phases. Such algorithms are said to be *synchronous*, in contrast with *asynchronous* algorithms, for which there is no notion of phases and the coordination of the computation of different processors is less strict.

Synchronous algorithms

A synchronous algorithm is mathematically equivalent to an algorithm governed by a global clock, that is, one for which 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 synchronous algorithm in an inherently asynchronous distributed system requires a *synchronization mechanism*, that is, an algorithm that is superimposed on the original and by which every processor can detect the end of each phase. Such algorithm is called a *synchronizer*. There are two main approaches on which synchronizers are based: *global synchronization* and *local synchronization*.

In global synchronization, the idea is to let each processor detect when *all* messages (packets of information) sent during a phase have been received, and only then start the computation of the next phase, whereas in local synchronization, 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. Under these conditions, it can be shown (Bertsekas, 1983) that the local synchronization method leads to no more communication penalty than any global synchronization method. The communication penalty often is considerably less. This is particularly the case when the transmission time of messages on communication links is comparable to the time needed for computation in each phase, since for global synchronization, additional messages may be required for acknowledgements, and so on. This suggests that local synchronization is superior to global synchronization in terms of communication penalty. There are other factors, however, such as software complexity, that support the global method.

Asynchronous algorithms

Consider a distributed algorithm. For each processor, there is a set of times at which the processor executes some computation, some other times at which it sends messages to other processors, and still others in which the processor receives messages from other processors. The algorithm is said to be *synchronous*, as mentioned earlier, if it is mathematically equivalent to one for which the times of computation, message transmission and message reception are fixed and established *a priori*. The algorithm is said to be *asynchronous* if these times and thus also the order of computations and message receptions at the processors can vary widely upon execution. The most extreme type of asynchronous algorithm is one that can tolerate changes in the problem data or in the distributed computing system, without restarting itself to some predetermined initial conditions.

There are several potential advantages that may be gained from asynchronous execution.

Implementation Flexibility. There is no overhead such as the one associated with global synchronization. Furthermore, in certain cases, there even are advantages over local synchronization methods.

Reduction of the Effects of Bottlenecks. Suppose that the computational power of processor i suddenly deteriorates drastically. In a synchronous execution, the entire algorithm would be slowed down. In an asynchronous execution, however, only the progress of $\boldsymbol{x}_i$ and all those components strongly influenced by $\boldsymbol{x}_i$ would be affected. The same argument applies to the case where a particular communication channel is suddenly slowed down.

Ease of Restarting. Suppose that the processors are engaged in the solution of an optimization problem and that suddenly one of the parameters of the problem changes. In a synchronous execution, all processors should be informed, abort the computation, and then restart (in a synchronized manner) the algorithm. In an asynchronous implementation no such re-initialization is required. Each processor incorporates the new parameter as soon as it learns the new value, without waiting for all other processors to become aware of the change. When all the processors learn the new parameter value, the algorithm becomes the correct (asynchronous) iteration.

Convergence Acceleration Due to Gauss-Seidel Effect. In the iterative solution of linear systems, Gauss-Seidel executions, usually converge with fewer updates of components, the reason being that new information is incorporated faster in the updated formulas. Asynchronous algorithms have the potential of displaying a Gauss-Seidel effect because the newest information is incorporated into the computations as soon as it becomes available, while retaining the maximal parallelism of a Jacobi-type algorithm.

Boundary Integral Formulation

Consider a system of N rigid particles immersed in a viscous fluid that itself may be undergoing an ambient flow described by $v^{\infty}(x)$. Each particle (label $\alpha = 1,\ldots,N$) is subject to a known external force, $\boldsymbol{F}^e_\alpha$ and torque $\boldsymbol{T}^e_\alpha$. As mentioned earlier, the challenge that takes this problem beyond the particle-particle interaction problems of molecular dynamics is that we must solve for not only the unknown motions of the particle, but also the motion of the intervening fluid. The motion of an incompressible Newtonian fluid of viscosity, in situations where inertial effects may be neglected, are governed by the Stokes equations:

$$-\nabla p+\mu\nabla^2 v=0, \quad \nabla\cdot v=0. \tag{7}$$

Here p is the pressure field and μ is the viscosity of the fluid. The solenoidal condition is the usual statement of the continuity equation for an incompressible fluid.

CDL-BIEM: contraction mappings and iterative solution

In an earlier series of works (Karrila et al., 1989; Kim and Karrila, 1991; Pakdel and Kim, 1991) we showed that the above three-dimensional partial differential equation can be satisfied by a velocity representation of the form:

$$v(x)-v^{\infty}(x)=v^{RC}(x)+\oint_S K(x-\xi)\cdot\varphi(\xi)dS(\xi), \tag{8}$$

The kernel of the integral operator is given by (We introduce the factor of 2 in the definition of the kernel so that the eigenvalues of the operator will scale between ± 1.):

$$K(x-\xi)=-2n(\xi)\cdot\Sigma(x-\xi), \tag{9}$$

where

$$\Sigma(x)=-\frac{3}{4\pi}\frac{xxx}{|x|^5}, \tag{10}$$

is the stress field of the Green's dyadic for the Stokes equation. The Green's dyadic may be written as $\mathcal{G}(x)/8\pi\mu$, where

$$\mathcal{G}_{ij}(x)=\frac{1}{|x|}\delta_{ij}+\frac{1}{|x|^3}x_i x_j, \tag{11}$$

is also known as the Oseen tensor. The integral operator involves spatial derivatives of the fundamental solution and in fact represents a surface distribution of sources/sinks and force dipoles. In analogy with a similar result from potential theory, the integral term is called the (hydrodynamic) double-layer potential (Odqvist, 1930). The unknown surface function $\varphi(\xi)$ is the double-layer density.

The velocity field $v^{RC}(x)$ which we call the range completer, is specified *a priori* and its role can be explained as follows. The double-layer potential, being a distribution of sources, sinks and force dipoles, cannot represent by itself, particle disturbance velocity fields in which forces or torques are exerted between the particle and the fluid. As its name implies, the range completer is a solution of the Stokes equation that imparts a net force and torque to the fluid, thus completing the range of the integral operator. The simplest example of a range completer is the linear combination of a point force and point torque velocity fields emanating from points inside each particle:

$$v^{RC}(x)=\sum_{\alpha=1}^{N}\left[F^e_\alpha-\frac{1}{2}(T^e_\alpha\times\nabla)\right]\cdot\frac{\mathcal{G}(x-x_\alpha)}{8\pi\mu}. \tag{12}$$

For particles of complex shape, instead of placing the Stokes singularities at one point per particle as in Eq. 12, a more elaborate distribution based, for example, on results from a rough calculation may be employed resulting in smoother solutions for φ, as shown by Pakdel and Kim (1991). Smoother solutions permit the use of coarser meshes and thus lead to fewer computations.

The unknown double-layer density is obtained by applying the boundary condition: on the boundaries of our fluid domain (for example, on S_α, the particle surface) we have the usual no-slip boundary condition, $v=U_\alpha+\omega_\alpha\times(x-x_\alpha)$, so that

$$U_\alpha+\omega_\alpha\times(x-x_\alpha)=v^{\infty}+v^{RC}+\varphi(x)+\mathcal{K}(\varphi), \quad x\in S_\alpha, \tag{13}$$

where the operator notation,

$$\mathcal{K}(\varphi)=\oint_S K(x,\xi)\cdot\varphi(\xi)dS(\xi),$$

has been used for the double-layer potential. Note that in Eq. 13 φ also appears outside the integral, thus yielding a *Fredholm integral equation of the second kind*. The velocity field produced by a double-layer distribution has a jump of strength $2\varphi(\xi)$ across the surface at ξ, and the term appearing outside the integral follows from replacing the limit of the surface integral, as the surface is approached, by the integral evaluated on the surface, the difference being exactly $\varphi(\xi)$ or one-half of the total jump.

To complete the set of equations, the unknown particle rigid-body motion, $U_\alpha+\omega_\alpha\times(x-x_\alpha)$ is related to the double-layer density by the exact result (Karrila et al., 1989; Kim and Karrila, 1991)

$$-U_\alpha=\sum_{i=1}^{3}\varphi^{(i,\alpha)}\oint_{S_\alpha}\varphi^{(i,\alpha)}\cdot\varphi dS, \tag{14}$$

$$-\omega_\alpha\times(x-x_\alpha)=\sum_{i=4}^{6}\varphi^{(i,\alpha)}\oint_{S_\alpha}\varphi^{(i,\alpha)}\cdot\varphi dS, \tag{15}$$

where the functions ($\varphi^{(i,\alpha)}$, $i=1,2,\ldots,6$, $\alpha=1,2,\ldots,N$ correspond to the six rigid-body motions defined on each of N particle surfaces. (The *support* of $\varphi^{(i,\alpha)}$ is the surface of particle α):

$\varphi^{(1,\alpha)}=\dfrac{e_1}{\sqrt{S_\alpha}}$, RBM translation in the x-direction

$\varphi^{(2,\alpha)}=\dfrac{e_2}{\sqrt{S_\alpha}}$, RBM translation in the y-direction

$\varphi^{(3,\alpha)}=\dfrac{e_3}{\sqrt{S_\alpha}}$, RBM translation in the z-direction

$\varphi^{(4,\alpha)}=\dfrac{\xi_2 e_3-\xi_3 e_2}{\sqrt{I_1}}$, RBM rotation about the x-axis

$\varphi^{(5,\alpha)}=\dfrac{\xi_3 e_1-\xi_1 e_3}{\sqrt{I_2}}$, RBM rotation about the y-axis

$\varphi^{(6,\alpha)}=\dfrac{\xi_1 e_2-\xi_2 e_1}{\sqrt{I_3}}$. RBM rotation about the z-axis

Here, the origin and directions of the coordinate axes must be taken as the "particle shell" center of mass and the principal directions of the "particle shell" moment of inertia tensor, S_α is the particle surface area, and I_1, I_2, and I_3 are the moments of inertia about the principal axes,

$$I_1=\oint_{S_\alpha}(\xi_2^2+\xi_3^2)dS,\quad I_2=\oint_{S_\alpha}(\xi_3^2+\xi_1^2)dS,$$

$$I_3=\oint_{S_\alpha}(\xi_1^2+\xi_2^2)dS.$$

The dependence on α is omitted for brevity in the notation.

There are two important points to be made here:

1. The $6N\varphi^{(i,\alpha)}$ form an orthonormal basis of the eigenspace $\lambda=-1$ of the operator $\mathcal{K}$. The inner product is the so-called natural inner product, which is defined as the integral of the usual vector dot product over the boundaries of the fluid domain (all particles surfaces). The orthonormal relations may be written explicity as:

$$\langle\varphi^{(i,\mu)},\varphi^{(j,\nu)}\rangle=\oint_S\varphi^{(i,\mu)}\cdot\varphi^{(j,\nu)}dS=\delta_{ij}\delta_{\mu\nu}.$$

2. The proof that the particle velocities are given by Eq. 14 and 15 is nontrivial and quite lengthy. For more details see Chapters 15-17 of Kim and Karrila (1991).

Substitution of Eq. 14 and 15 into Eq. 13 yields the final form for the boundary integral equation which is to be solved to determine the double-layer density φ:

$$\sum_{\alpha=1}^{N}\sum_{i=1}^{6}\varphi^{(i,\alpha)}(x)\oint_{S_\alpha}\varphi^{(i,\alpha)}\cdot\varphi dS+\varphi(x)+\mathcal{K}(\varphi)$$
$$=-v^\infty(x)-v^{RC}(x),\quad \text{for } x\in S_\alpha. \tag{16}$$

The form of this equation suggests an iterative solution algorithm. Upon discretization (collocation and integration by quadratures), the integral equation becomes a large linear system of the form $x=Mx+b$, where the vector x contains the discretized information on φ and the matrix M contains information originating from the first and third terms on the lefthand side of Eq. 16. The iterative solution method,

$$x^{(n+1)}=Mx^n+b, \tag{17}$$

converges to the fixed point, if and only if the eigenvalues of M have norm less than one. If this condition is met, the rate of convergence is dictated by the largest eigenvalue. With each successive iteration, the "distance" of approximate solution x_n from the actual solution will diminish by at least a factor equal to the norm of this dominant eigenvalue.

Now the eigenvalues of $\mathcal{K}$ are of the form (Odqvist, 1930, contains a sketch of the proof; for the complete steps, see Kim and Karrila, 1991):

$$\lambda=\frac{E^{(i)}-E^{(o)}}{E^{(i)}+E^{(o)}} \tag{18}$$

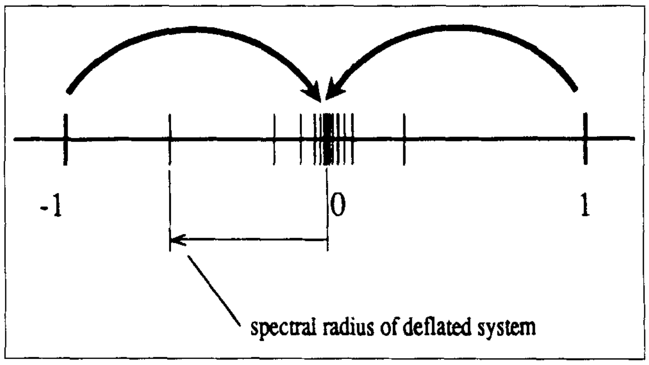

Figure 3. Wielandt deflation of the end points of the double layer eigensystem.

where $E^{(i)}\geq 0$ and $E^{(o)}\geq 0$ are the energy dissipation rates inside and outside the surfaces, of the velocity field produced by the eigenfunction acting as a double-layer density. So λ must lie on the real line between -1 and 1 (inclusive), with the end points of the spectrum corresponding to the case with zero velocity in the inside or outside regions. So if we rewrite Eq. 16 in operator notation as:

$$(1+\mathcal{H})\varphi=-(v^\infty+v^{RC}),$$

the governing equation involves an integral operator that is nothing more than a compact perturbation of the identity, with $\mathcal{H}$ obtained from a Wielandt deflation of $\mathcal{K}$:

$$\mathcal{H}=\mathcal{K}+\sum_{i,\alpha}\varphi^{(i,\alpha)}\langle\varphi^{(i,\alpha)},.\rangle. \tag{19}$$

Each $\varphi^{(i,\alpha)}$ is also an eigenfunction of $\mathcal{H}$, but with eigenvalue 0 (hence the name deflation) instead of -1.

We now exploit the fact that the set of eigenvalues (spectrum) of the double-layer operator is discrete, with an accumulation point at the origin, consistent with the spectral theory for compact operators (Friedman, 1982; Ramkrishna and Amundson, 1985). A typical spectrum for the double-layer operator $\mathcal{K}$ is shown in Figure 3; it is now obvious that if we deflate the eigenvalues at $+1$, an iterative approach would be successful, since the discrete nature of the spectrum guarantees that after deflation, the outermost eigenvalue would be inside the interval $(-1, 1)$.

Unfortunately, at $\lambda=+1$, the eigenfunctions of $\mathcal{K}$ depend on the particle shape and are therefore not known *a priori*, so we cannot project onto those functions to achieve the deflation. However, the eigenfunctions of its adjoint, $\mathcal{K}^*$, are known. For an N-particle system, the eigenspace at $\lambda=+1$ is of dimension N; the N particle surface normals, $n_\alpha(\xi)$, with the appropriate normalization by surface area,

$$\psi_\alpha(\xi)=\frac{n_\alpha(\xi)}{\sqrt{S_\alpha}},$$

form an orthonormal basis. (The support for $n_\alpha(\xi)$ and ψ_α is S_α.) Projections using these eigenfunctions achieve the desired deflations at $+1$. The details are given by Kim and Karrila

(1991), but the final results are as follows. We define a projection operator $\mathcal{P}$ by:

$$\mathcal{P}\varphi = \sum_{\alpha=1}^{N} \psi_\alpha < \varphi, \psi_\alpha > .$$

Then the following problems have the same solution:

1. $(1+\mathcal{K})\varphi = u$,

2. $(1+\mathcal{K}-\mathcal{P})\varphi = \left(1-\frac{1}{2}\mathcal{P}\right)u$.

In problem 2, the eigenvalues of $\mathcal{K}-\mathcal{P}$ all lie inside the interval $(-1, 1)$. Thus the results presented in this work were obtained by applying the usual boundary element discretization methodologies to:

$$\varphi^{(n+1)} = -(\mathcal{K}-\mathcal{P})\varphi^{(n)} - \left(1-\frac{1}{2}\mathcal{P}\right)(v^{\infty}+v^{RC}). \quad (20)$$

or

$$\varphi^{(n+1)} = -\tilde{\mathcal{K}}\varphi^{(n)} + b. \quad (21)$$

The end result is a large, dense system of equations. Since the essence of the solution procedure consists of matrix construction and matrix-vector multiplication, fairly large computations (on the order of 200,000 boundary elements) have been implemented on conventional super-minicomputers (Phan-Thien et al., 1992). However, the parallelism and fine granularity in the algorithm is quite apparent (see Figure 4) since elements in the product vector can be calculated independently with the workload balanced evenly over multiple processors.

Iterative solution of linear systems

Iterative solution of linear systems is a special case of the algorithm:

$$x = f(x), \quad (22)$$

where $x(t)$ is a vector in $\mathbb{R}^n$ and f: $\mathbb{R}^n \rightarrow \mathbb{R}^n$ is an iteration mapping defining the algorithm. These are called *iterative* algorithms or, in certain contexts, *relaxation* methods. A special case of particular interest to us arises when the function f is of the form $f(x) = \tilde{\mathcal{K}}x + b$, where $\tilde{\mathcal{K}}$ is a square matrix and b is a vector, in which case we are dealing with a *linear* iterative algorithm.

Often we have to use a coarse-grained parallelization of Eq. 22. We can decompose the vector space $\mathbb{R}^n$ into a Cartesian product of lower dimensional subspaces $\mathbb{R}^{n_j}, j = 1, \ldots, p$, where $n = n_1 + \ldots + n_p$. Accordingly, any vector $x \in \mathbb{R}^n$ can be expressed in the form $x = (x_1, \ldots, x_p)$, with each $x_j \in \mathbb{R}^{n_j}$. The iteration scheme can be written as:

$$x_j = f_j(x), \quad j = 1, \ldots, p \quad (23)$$

where each f_j is a vector function mapping $\mathbb{R}^n$ to $\mathbb{R}^{n_j}$. The

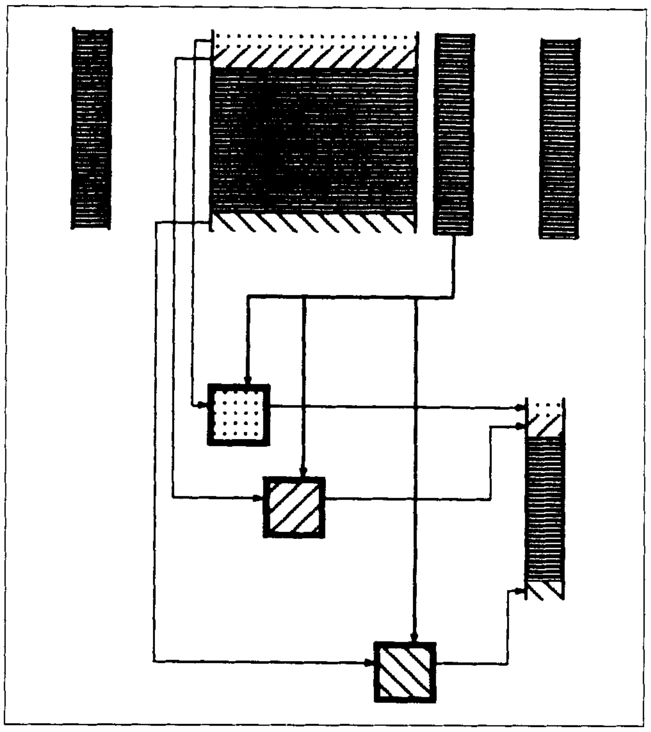

Figure 4. The parallel algorithm for iterative solution of the linear system $x = Mx + b$.

resulting algorithm is said to be *block-parallelized*. There are several reasons for being interested in block-parallelization. First, there may be too few processors available, so that we have to assign more than one component to each one of them. Second, certain functions f_j may involve common computations, and thus it would be natural to group them together.

Synchronous iterations

We say that an execution of the iteration, Eq. 22, is synchronous if it can be described mathematically by:

$$x(t+1) = f(x(t)), \quad (24)$$

where t is an integer-valued variable used to index different iterations, not necessarily representing time. Equation 24, in which all components of x are updated simultaneously, is sometimes called a *Jacobi*-type iteration. In an alternative form, the components of x are updated one at a time, and the most recently computed values of the components are used. The resulting iteration is often called an iteration of the *Gauss-Seidel*-type and can be expressed mathematically as:

$$x_i(t+1) = f_i(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_n(t)),$$
$$i = 1, \ldots, n. \quad (25)$$

Gauss-Seidel algorithms are often preferable: they incorporate the newest information and for this reason, they generally converge faster than the corresponding Jacobi algorithms. In fact, under certain conditions Jacobi iterations fail to converge while Gauss-Seidel iterations are guaranteed to converge. However, the parallel implementation of the iteration for Eq. 24

can be problematic. If no two components can be updated in parallel (every component depends on every other component), the Gauss-Seidel iteration will be completely nonparallelizable. On the other hand, if the dependencies are weaker, Gauss-Seidel iterations can be substantially parallelizable.

Asynchronous iterations

Asynchronous iterations for the solution of linear systems of equations were introduced by Chazan and Miranker (1969), under the name of *chaotic relaxation*. As discussed before, in an asynchronous implementation of Eq. 22, processors are not required to wait to receive all messages generated during the previous iteration. Furthermore, processors are not required to communicate their results after every iteration but only once in a while. We allow some processors to compute faster and execute more iterations than others, we allow some processors to communicate more frequently than others, and we allow communication delays to be substantial and unpredictable and communication channels to deliver messages out of order, not in the order in which they were transmitted.

A major potential drawback of asynchronous algorithms is that they cannot be modeled easily. However, it is possible to formulate an algorithmic model for the asynchronous implementation of the iteration, Eq. 22 (Bertsekas, 1983). We conclude this section with a description of the asynchronous model and present a general convergence theorem that will be fundamental in the successful implementation of parallel computational strategies.

Consider the sets $X_1, X_2, \ldots, X_n$ and their Cartesian product:

$$X = X_1 \times X_2 \times \ldots \times X_n.$$

The elements of X can be written as:

$$x = (x_1, x_2, \ldots, x_n) \quad x \epsilon X$$

where the x_i are the corresponding elements of X_i. Let f_i: $X \rightarrow X_i$ be given functions, and let f: $X \rightarrow X$ be the function defined by:

$$f(x) = f_1(x), f_2(x), \ldots, f_n(x), \text{ for all } x \epsilon X.$$

The problem is to find a fixed point of f, that is, an element $x^* \epsilon X$ such that $x^* = f(x^*)$, or in terms of the components,

$$x_i^* = f_i(x^*) \quad i = 1,2,\ldots,n.$$

This analysis is fairly general, the only implicit assumption used so far is that the notion of convergence is defined on X. We will now describe a distributed asynchronous version of the iterative method:

$$x_i = f_i(x), \quad i = 1,2,\ldots,n.$$

Let

$$x_i(t) = \text{value of } i\text{th component at time } t.$$

Let us assume that there is a set of times $T = \{0,1,2,\ldots\}$ at which one or more components (x_i) of x are updated by some processor. The elements of T should be considered as the indices of the sequence of physical times at which the updates take place. Let

$$T^i = \text{set of times at which } x_i \text{ is updated.}$$

We have to assume that the processor updating x_i may not have access to the most recent value of the components of x, that is, we assume that:

$$x_i(t+1) = \begin{cases} f_i(x_1(\tau_1^i(t), \ldots, x_n(\tau_n^i))), & \text{for all } t \epsilon T^i, \\ x_i(t) & \text{for all } t \notin T^i. \end{cases} \tag{26}$$

where $\tau_j^i(t)$ denotes the time at which the jth component becomes available at the processor updating $x_i(t)$ and satisfies:

$$0 \leq \tau_j^i(t) \leq t \quad \text{for all } t \epsilon T.$$

The sets T^i, as well as the sequences of physical times that they represent, need not be known to any one processor, since the knowledge is not required to execute the iteration in Eq. 26, that is, there is no requirement for a shared global clock or synchronized local clocks at the processors.

We will assume that the sets T^i are infinite, and if $\{t_k\}$ is a sequence of elements of T^i that tends to infinity, then $\lim_{k \rightarrow \infty} \tau_j^i(t_k) = \infty$ for every j. This assumption (*total synchronism*) guarantees that each component is updated infinitely often and that old information is eventually purged from the system.

We will also assume that there is a sequence on nonempty sets $\{X(k)\}$ with

$$\ldots \subset X(k+1) \subset X(k) \subset \ldots \subset X$$

satisfying the following two conditions:

Synchronous Convergence Condition. We have:

$$f(x) \epsilon X(k+1), \text{ for all } k \text{ and } x \epsilon X(k).$$

Box Condition. For every k, there exist sets $X_i(k) \subset X_i$ such that:

$$X(k) = X_1(k) \times X_2(k) \times \ldots \times X_n(k).$$

The first condition implies that the limit points of sequences generated by the (synchronous) iteration $x = f(x)$ are fixed points of f, assuming that the initial x belongs to $X(0)$. The second condition implies that by combining components of the vectors in $X(k)$, we obtain vectors in $X(k)$.

We can now state the *asynchronous convergence theorem* (Bertsekas, 1983). Under the assumption of total synchronism, if the synchronous convergence and box conditions hold, and the initial solution estimate,

$$x(0) = [x_1(0), \ldots, x_n(0)],$$

belongs to the set $X(0)$, then every limit point of $\{x(t)\}$ is a fixed point of f.

We can apply this important result to the solution of linear systems of equations. Consider the case where

$$f(x) = \mathcal{H}x + b$$

where $\mathcal{H}$ is an $n \times n$ matrix and b a vector in $\mathbb{R}^n$. Therefore, we want to find x^* such that:

$$x^* = \mathcal{H}x^* + b$$

using an asynchronous version of the iteration $x = \mathcal{H}x + b$ given by:

$$x_i(t+1) = \begin{cases} \sum_{j=1}^{n} h_{ij} x_j(\tau_j^i(t)) + b_i, & \text{for all } t \in T^i \\ x_i(t) & \text{for all } t \notin T^i. \end{cases}$$

where the h_{ij}'s are the components of $\mathcal{H}$. In this case, it can be shown (Bertsekas, 1983) that the asynchronous convergence theorem applies, provided:

$$\rho(|\mathcal{H}|) < 1$$

where $\rho(|\mathcal{H}|)$ is the spectral radius of the matrix $|\mathcal{H}|$ having as elements the absolute values $|h_{ij}|$.

Communication Scheduling Strategies

Introduction

The CDL-BIEM algorithm is naturally parallelizable, but the programmer still has latitude in mapping the algorithm to a given computer architecture. Depending on the floating point performance and memory capacity of the processors that make up the computer, we may choose, for example, to assign a processor to each particle. Very good results have been obtained for small-scale systems [$O(10)$ particles], with errors on the order of 1% with respect to analytical or experimental results (Karrila et al., 1989). However, if we move to larger systems [$O(10^{2-3})$ particles] it would be practically impossible to implement directly, the algorithms that have been used so far. The problem, of course, is not related to the solution approach but to limitations associated with current computer architectures.

There are two aspects of computer architecture which play a deciding role in successful scale-up of a parallel algorithm: *memory availability* and *interprocessor communications*.

We have developed algorithms that get around the problem of *memory* requirements, by partial recalculation of the iteration matrix. Furthermore, at the end of this article, we illustrate how recalculation of the iteration matrix is kept to a minimum in algorithms that are currently being developed for large-scale simulations. It turns out that most portions of the matrix, that is, the off-diagonal blocks corresponding to interactions between distant particles, are needed only infrequently in larger simulations.

Let us now consider the communication issue. A synchronous implementation of the Jacobi iteration algorithm requires each processor to have access to the previous iteration vector. In a shared memory system such as the sequent symmetry this is readily accomplished from a programming point of view (Karrila et al., 1989). But with larger solution vectors there will be bus contention in accessing of data. In a distributed memory system, the processors have to communicate with one another, requesting and sending data to keep updated information. The communication load scales as N^2, where N is the number of processors used. As the number of particles (processors) increases, interprocessor communication becomes the limiting phase in the solution of the problem, eventually reaching a state in which the processors are busy exchanging information and the iterative procedure is stalled.

For large-scale simulations, any modification of the iterative algorithm that reduces the communication loads will lead to a significant speedup in the solution of the problem. We have found that using physical insights into the nature of double-layer hydrodynamic interactions, it is possible to determine approximately how often the processors need to communicate (exchange information). The resulting asynchronous algorithm forces the processors to work with partially outdated information. However, it can be proven rigorously that since all processors eventually update required information, the algorithm is convergent.

Scheduling algorithms

Hydrodynamic double-layer interactions are weak, short-range interactions: the strongest double-layer potential corresponds at most to a stresslet (symmetric force dipole) field, and for particles of high symmetry, the spectral radius (norm of the largest eigenvalue) is associated with an eigenfunction double-layer potential that generates the even weaker Stokes quadrupole field. These weak interactions can be directly exploited for cluster simulations and dilute suspension theory, by requiring the frequency of interprocessor communications to be dictated by the distance between the particles. That is, if the particles are close to each other and interact strongly (in the hydrodynamic sense), the associated processors should exchange information very often; if the particles are far apart and interactions are weak, the corresponding processors would not need to communicate as often.

Explicit association rules, which we call *communications scheduling strategies*, will be derived to determine how often processors should exchange information. The two-particle problem will provide the basis for the scheduling strategy; although from a computational point of view it can be considered a *pairwise established* strategy, we will show that it forms the basis for a successful scheduling of the complete N-body hydrodynamic interaction problem.

Two-sphere problem

Consider two equal spheres with their centers separated by a distance R. Let $\{P_1, P_2\}$ be the processors assigned to spheres $\{S_1, S_2\}$, respectively. Given the forces and torques acting on the particles, that is, a mobility problem, CDL-BIEM leads to an iterative scheme that can be written in the form (see Eq. 21):

$$\begin{pmatrix} \varphi_1^{(k+1)} \\ \varphi_2^{(k+1)} \end{pmatrix} = \begin{pmatrix} \mathcal{H}_{11} & \mathcal{H}_{12} \\ \mathcal{H}_{21} & \mathcal{H}_{22} \end{pmatrix} \begin{pmatrix} \varphi_1^{(k)} \\ \varphi_2^{(k)} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \qquad (27)$$

$$k = 0, 1, 2, \ldots,$$

where the interaction matrix $\tilde{\mathcal{K}}$, the load vector b, and the iteration vector φ have been partitioned explicitly to indicate the action of each processor (denoted by the corresponding subscript) in the iterative procedure.

We now allow each processor to iterate locally on the corresponding portion of the iteration vector, without access to the information generated by the iterations on the other processor. This decoupled iteration scheme is of the form:

$$\begin{aligned}(\varphi_1^{(k'+1)}) &= (\tilde{\mathcal{K}}_{11}) \quad (\varphi_1^{(k')}) + (\tilde{\mathcal{K}}_{12}\varphi_2^{(k)} + b_1)\\ (\varphi^{(k'+1)}) &= (\tilde{\mathcal{K}}_{22}) \quad (\varphi_2^{(k')}) + (\tilde{\mathcal{K}}_{21}\varphi_1^{(k)} + b_2 \\ k' &= 0, 1, 2, \ldots .\end{aligned} \tag{28}$$

with

$$\begin{aligned}(\varphi_1^{(k')})_{k'=0} &= (\varphi_1^{(k)})\\ (\varphi_2^{(k')})_{k'=0} &= (\varphi_2^{(k)})\end{aligned}$$

If we allow both processors to perform *n subiterations* ($k' = 0, 1, \ldots, n-1$) before they exchange information to update the iteration vector, it is not difficult to show that the *effective* iteration scheme can written as:

$$\begin{pmatrix}\varphi_1^{(k+1)}\\ \varphi_2^{(k+1)}\end{pmatrix} = \begin{pmatrix}\tilde{\mathcal{K}}_{11}^n & \Sigma_{j=0}^{n-1}\tilde{\mathcal{K}}_{11}^j\tilde{\mathcal{K}}_{12}\\ \Sigma_{j=0}^{n-1}\tilde{\mathcal{K}}_{22}^j\tilde{\mathcal{K}}_{21} & \tilde{\mathcal{K}}_{22}^n\end{pmatrix}\begin{pmatrix}\varphi_1^{(k)}\\ \varphi_2^{(k)}\end{pmatrix} + \begin{pmatrix}\sum_{j=0}^{n-1}\tilde{\mathcal{K}}_{11}^j b_1\\ \sum_{j=0}^{n-1}\tilde{\mathcal{K}}_{22}^j b_2\end{pmatrix}, \quad k = 0, 1, 2, \ldots . \tag{29}$$

As expected if $n = 1$, that is, if the processors exchange information after every iteration, Eq. 29 reduces to the original Jacobi iteration scheme (Eq. 27). We will refer to $m = (n-1)$ as the *delay* in the iteration scheme.

The delayed iteration scheme (Eq. 29) does not provide a practical method to solve the mobility problem because it is computationally expensive. However, the closed form expression for the effective iteration matrix can be used to carry out a numerical study on the convergence of the delayed scheme.

The behavior of the ratio:

$$r_m \doteq \frac{\rho_m}{\rho_0} = \frac{\text{spectral radius of delayed } (m) \text{ scheme}}{\text{spectral radius of undelayed scheme}}$$

as a function of the gap (δ) between the spheres was explored numerically (Figure 5), and we found that:

1. For all $\delta > 0$, $r_m < 1$.
2. Retaining the accuracy of the results as the gap goes to zero requires finer discretizations of the particle surfaces, therefore we can only infer that for all m,

$$\lim_{\delta \to 0} r_m = 1.$$

3. For all $\delta > 0$ and $m > m'$, $r_m < r_{m'}$.
4. If we denote by ρ_{ss} the spectral radius of the single-sphere problem, then for all m,

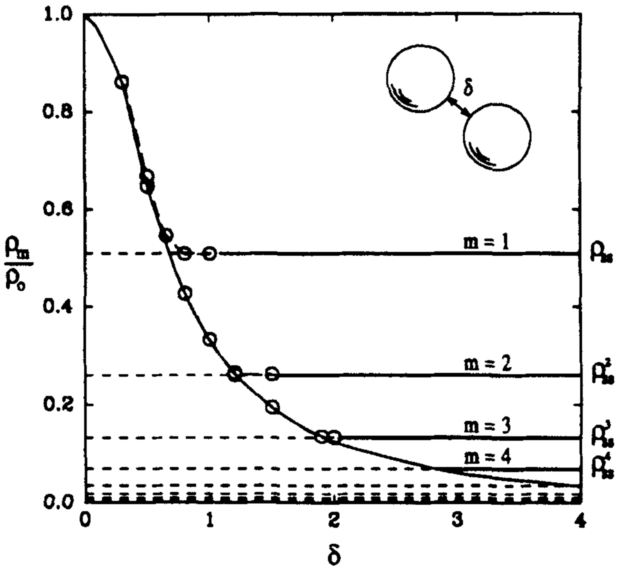

Figure 5. Spectral radius reduction, 2-sphere problem: ratio of spectral radii delayed by *m* to undelayed, vs. gap between spheres.

$$\lim_{\delta \to \infty} r_m = \rho_{ss}^m.$$

5. For all $\delta > 0$, there exists a unique

$$r_\infty = \lim_{m \to \infty} r_m.$$

The curve $r_\infty(\delta)$ defines a spectral radius reduction *envelope*.

6. For all practical purposes, for each curve $r_m(\delta)$ there exists a δ_m, such that:

$$r_m(\delta) \approx \begin{cases} r_\infty(\delta) & \text{for } \delta < \delta_m, \\ \rho_{ss}^m & \text{for } \delta \geq \delta_m. \end{cases} \quad m = 0, 1, 2, \ldots .$$

Conceptually, for a two-sphere problem, Figure 5 contains all the information regarding the convergence of a delayed iteration scheme. In particular, for a given configuration, the set of bounds $\mathcal{B} = \{2 + \delta_m\}_{m=0}^{\infty}$ determines the minimum delay required to maximize the spectral reduction. The spectral radius of the delayed scheme, ρ_m, would dictate the rate of convergence of the iterative algorithm for a theoretical computer, in which the computations (subiterations) are performed infinitely fast (infinite communications penalty), and the convergence is strictly governed by the communication speed between the processors. Since real computers have finite communication penalties, ρ_m provides a lower bound for the observable rate of convergence (the observed spectral reduction).

The set $\mathcal{B}$ was constructed for two equal spheres; however, the procedure is valid for any two identical particles of arbitrary shape. In a more general problem involving particles of the same shape but different size, we can expect to have different delays (m_1, m_2) associated with the respective processors. However, preliminary studies with unequal spheres indicate that if the particles are not too close to each other, that is, as long as the same number of elements can be used to discretize the surface of both spheres, the spectral reduction is dictated by $m = \max\{m_1, m_2\}$ and we can recover the symmetric information exchange algorithm by letting $m_1 = m_2 = m$.

Spectral communication scheduling strategy

The results of the previous section particularly the existence of the set of bounds $\mathcal{B}$, provide the basis for the formulation of the *spectral communications scheduling strategy* for the N-particle problem.

Consider a mobility problem involving a set of N identical particles $\{S_i\}_{i=1}^N$. Let $\{P_j\}_{j=1}^N$ be the associated set of processors, then m_{ij}, the suggested delay in the information exchange between processors, P_i and P_j, is given by:

$$m_{ij} = \min_{m \in \mathfrak{N}} \{\delta_m \in \mathcal{B} \mid \delta_m \geq d_{ij}\} \quad i, j = 1, 2, \ldots, N \qquad (30)$$

where $\mathcal{B}$ is the set of bounds (based on the interparticle distance) of the two-particle problem, and d_{ij} is the distance between particles S_i and S_j.

The matrix $N = (m_{ij} + 1)$ specifies how often any two given processors should exchange information and is said to define a *schedule* for the solution algorithm. The convergence of the algorithm is guaranteed by the asynchronous convergence theorem (Bertsekas and Tsitsiklis, 1989) as long as the m_{ij}'s are finite.

Let $\mathfrak{N}$ denote the space of all possible schedules for a given problem. Given two schedules $N_1, N_2 \in \mathfrak{N}$, we will say that *schedule N_1 is better than N_2* if (and only if), as the iteration time goes to infinity, $\epsilon_1(t) < \epsilon_2(t)$, where $\epsilon_i(t)$ denotes the error in the solution (at time t) when schedule N_i is used, and we will write:

$$N_1 < N_2.$$

This definition leads naturally to the concept of an *optimal* schedule, that is, a schedule N^* such that:

$$N^* < N \text{ for all } N \in \mathfrak{N}.$$

The search for the optimal schedule is a challenging problem, unfortunately it is NP-complete (Manber, 1989). Therefore, we will not attempt to find a solution to this problem, instead we will study the performance of scheduling strategies and, based on this, reach some conclusions about their properties.

Stochastic scheduling

The spectral communications scheduling strategy presented in the previous section defines a *deterministic* strategy at the algorithmic level. In this section an alternative approach, stochastic in nature, will be introduced. As before, consider a mobility problem involving a set of N-identical particles $\{S_i\}_{i=1}^N$ and let $\{P_j\}_{i=1}^N$ be the associated set of processors.

Let E_{ij} be the event defined by:

E_{ij} = processor i sends updated information to processor j

and p_{ij} the probability of such event. A matrix $P = (p_{ij})$ defines a *stochastic schedule*, if at the end of every iteration each

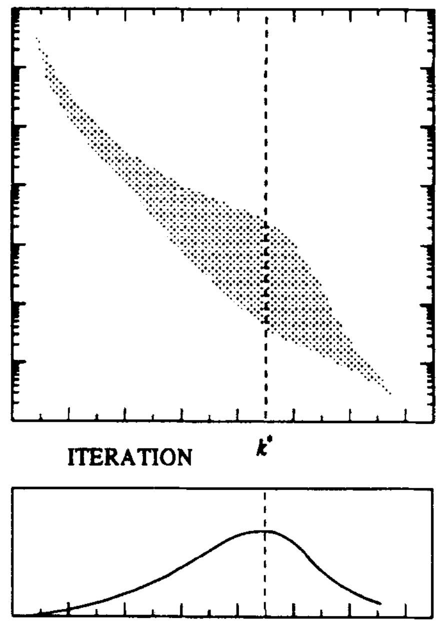

Figure 6. Stochastic schedule behavior.

The shaded region denotes the variance of the solution vector, vs. number of iterations.

processor decides whether to send the new information to other processors or not, based on this schedule. The asynchronous convergence theorem guarantees the convergence of an iterative algorithm using a stochastic schedule, as long as all the p_{ij}'s are nonzero.

Let $\mathcal{P}$ denote the space of all possible stochastic schedules and let X_k be the random variable denoting the state of the iteration vector after k iterations. If n is the number of $p_{ij} < 1$ in a schedule $P \in \mathcal{P}$, then the kth iteration induces 2^n possible states for the ($k+1$)th iteration, and thus the total number of possible states for X_k is 2^{kn}. Therefore, the range of X_k increases geometrically as the iterations proceed. The asynchronous convergence theorem, however, guarantees that the variance of X_k ($\mathrm{Var}(X_k)$) goes to zero as k goes to infinity. Since $\mathrm{Var}(X_k)|_{k=0} = 0$, we know that the variance attains a maximum for some $k^* < \infty$ (see Figure 6). We can conclude that, for all $P \in \mathcal{P}$, for all $\epsilon > 0$, there exists $k' > k^*$ such that $\mathrm{Var}(X_{k'}) < \epsilon$. Clearly, the location of the maximum k^* and the magnitude of the maximum $\mathrm{Var}(X_{k^*})$, both of which are inversely proportional to $\min\{p_{ij}\}$, are parameters that have to be considered in the implementation of a stochastic schedule.

We will conclude this section by discussing the construction of a stochastic schedule P. In general, the p_{ij} will be functions of the configuration of the particle system:

$$p_{ij} = p_{ij}(r_1, r_2, \ldots, r_N).$$

It is possible to postulate models based on the physical nature of the system being studied, for example:

$$p_{ij} = S(r_1, r_2, \ldots, r_N) f(|r_i - r_j|^{-\nu})$$

where S can account for screening phenomena in concentrated systems, and f can be tailored to account for the appropriate limiting behavior of these functions: at short distances the decay of the p_{ij}'s is dominated by the algebraic decay of f, whereas for large separations the screening effect introduced by S dominates the type of decay (for example, exponential).

These models can be simplified so as to make them a function only of the distance between the particles involved:

$$p_{ij} = p_{ij}(|r_i - r_j|).$$

We can put in this category the stochastic implementation of the spectral communications scheduling strategy: let N be the proposed (deterministic) schedule based on the set of bounds $\mathcal{B}$, then:

$$p_{ij} = \frac{1}{n_{ij}} = \frac{1}{m_{ij}+1}. \qquad (31)$$

Therefore, on average the communications of each processor follow the spectral communications schedule. It should be noted, however, that because we are not dealing with independent random variables, the mean behavior of this stochastic schedule will not, in general, coincide with that of the deterministic schedule.

Implementation of scheduling strategies on the Intel iPSC/860 supercomputer

The scheduling ideas discussed above should preferably be tested on a massively parallel computer featuring significant computational capabilities on each node. Since we are several years away from such machines, as a first step, the scheduling ideas were implemented on an Intel iPSC/860 computer featuring a hypercube interprocessor connection. In this section we will discuss important features of the programming of the spectral communications scheduling strategy, in both the deterministic and the stochastic versions. We will begin by briefly describing the main characteristics of the iPSC/860 (to record current resources in the context of future implementations).

The iPSC/860 is a parallel, modular and scalable supercomputer system based on Intel's i860 microprocessor and a proprietary communications network. The multiple-processing elements that make up the system, also known as *nodes*, are complete, self-contained computers with substantial execution speed (RISC technology) and local memory capacity.

This distributed memory hypercube can be configured with up to 128 nodes, each with up to 64 MBytes of memory, and a peak double-precision performance of 40 MFlops (10^6 floating point operations). Therefore, the system's aggregate performance can be scaled from 480 peak MFlops to 7.6 GFlops (10^9 floating point operations), with random access memory ranging up to 8 GBytes. The iPSC/860 available to us is configured with 16 nodes, with 16 MBytes of memory on each node.

The system has a small front end computer, the system resource manager (SRM), which also happens to be the default I/O processor. This machine, also known as the *host*, and the nodes are connected by a high-performance network, optimized for message-oriented communications, which automatically and transparently ties together all processors and provides a high-speed data pathway between them.

The iPSC/860 has a Unix-based software development environment which supports a multiuser programming team in a familiar workstation setting, while providing a set of tools for developing and debugging concurrent applications.

The implementations of the parallel strategies for the solution of the N-particle mobility problem on the iPSC/860 was as follows.

The Host. It was assigned three tasks:

1. The initialization, which included reading the input data files, discretizing the surface of the particles, and providing the nodes with all the common blocks and the initial guess required to perform the iterations.
2. The supervision of the iterative procedure: collecting the information sent by the nodes and checking the convergence of the solution.
3. The post-processing required to extract physical quantities (that is, translational and rotational velocities) from the solution given by the iterative algorithm and writing the output files.

All of these tasks are essentially sequential. They, however, take a very small fraction of the total execution times, thus allowing overall execution speedups that are close to the ideal limit.

The Nodes. The computational nodes (where the i860 processors reside) performed four basic tasks:

1. The initialization, which included receiving the required common blocks and initial guess from the host, calculating the portion of the interaction matrix corresponding to that node and storing it in local memory.
2. The actual iteration procedure, that is, the dot product of the interaction matrix with the iteration vector plus the corresponding portion of the load vector.
3. Following a given schedule, perform all required interprocessor communications asynchronously. The implementation of stochastic schedules also requires that the nodes take care of the initialization and use of a random number generator.
4. Asynchronously sending updated information to the host with the frequency of updating being supplied by the user.

The coding of this program was done entirely in FORTRAN, essentially the same one used in sequential computers, except, of course, for the calls to send/receive routines and other functionalities associated with distributed memory computing. The computational performance numbers, which appear in the following discussion, serve only as a conservative and illustrate the role of communication bottlenecks. For a program written in a high-level language, the ratio of computation to communication will increase with compiler performance. This is particularly relevant for the i860 processor in light of its short compiler history. After these runs, new FORTRAN compilers have arrived on the scene, and the ratio between sustained floating point speeds and communications has increased by a factor of 5 (or even more by using i860 machine codes).

Performance of scheduling strategies

We present a simple four-sphere problem to illustrate the performance of the different scheduling strategies. We believe this problem is large enough to capture most of the essential features of each strategy and represents the first step toward gaining a perspective on large-scale problems.

Consider four identical spheres of radius $r = 1$ settling under the influence of a constant force (for example, gravity). We take gravity in the negative z-direction and place all sphere centers on the xy-plane. Their positions (centers) are given by:

$$\mathbf{r}_1 = (0.00,\ 0.00)$$
$$\mathbf{r}_2 = (0.00,\ 3.40)$$
$$\mathbf{r}_3 = (0.00,\ 5.90)$$
$$\mathbf{r}_4 = (2.60,\ 0.00).$$

The associated distance matrix is:

$$\boldsymbol{D} = \begin{pmatrix} 0.00 & 3.40 & 5.90 & 2.60 \\ 3.40 & 0.00 & 2.50 & 4.28 \\ 5.90 & 2.50 & 0.00 & 6.45 \\ 2.60 & 4.28 & 6.45 & 0.00 \end{pmatrix}.$$

The first elements in the set of bounds for the two-sphere problem are (see Figure 5):

$$\boldsymbol{B} = \{2.00,\ 2.40,\ 3.20,\ 3.90,\ 4.97,\ 5.90,\ 7.47,\ 9.23,\ \ldots\}$$

therefore, the suggested delay matrix is:

$$\boldsymbol{M} = \begin{pmatrix} 0 & 3 & 5 & 1 \\ 3 & 0 & 1 & 4 \\ 5 & 1 & 0 & 6 \\ 1 & 4 & 6 & 0 \end{pmatrix}.$$

The spectral communications schedule is given by:

$$\boldsymbol{N}_{SCS} = \begin{pmatrix} 1 & 4 & 6 & 2 \\ 4 & 1 & 2 & 5 \\ 6 & 2 & 1 & 7 \\ 2 & 5 & 7 & 1 \end{pmatrix},$$

and therefore, the associated stochastic schedule is:

$$\boldsymbol{P}_{SCS} = \begin{pmatrix} 1 & 1/4 & 1/6 & 1/2 \\ 1/4 & 1 & 1/2 & 1/5 \\ 1/6 & 1/2 & 1 & 1/7 \\ 1/2 & 1/5 & 1/7 & 1 \end{pmatrix}.$$

These schedules will be compared with the performance of the regular point Jacobi iteration schedule, which can be expressed as:

$$\boldsymbol{N}_J = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

which is numerically identical to the associated stochastic schedule $\boldsymbol{P}_J$.

The surface of the spheres was discretized into a 80-element polyhedron (N_E), using the tessellation algorithm described by Karrila et al. (1989): Start with a platonic solid, for example, an icosahedron, subdivide triangles into four triangles, and project new vertices to the sphere surface. The performance was evaluated by studying the convergence rate of the resulting algorithm. We monitored the absolute error in the iterated vector φ, the double-layer density, as a function of the number of iterations. The *absolute error* at the kth iteration was defined as:

$$\epsilon_k = \frac{1}{3NN_E} \sum_{\alpha,i,j} \left| \frac{\varphi^k_{\alpha,i,j} - \varphi^\infty_{\alpha,i,j}}{\varphi^\infty_{\alpha,i,j}} \right| \tag{32}$$

$$\alpha = x,\ y,\ z, \quad i = 1,\ 2,\ \ldots,\ N, \quad j = 1,\ 2,\ \ldots,\ N_E,$$

where $\varphi^k_{\alpha,i,j}$ denotes the α-component of the double-layer density on jth element in the ith particle of the system, after k iterations. The *exact* solution to the problem, denoted by $\varphi^\infty_{\alpha,i,j}$, corresponds to the fully converged solution obtained, for example, by using the regular point Jacobi algorithm.

The convergence of the double-layer density is a most demanding performance measure. On the other hand, in practical applications, we would be interested in the physical quantities that can be extracted from the knowledge of the double-layer density such as particle motions. All of these quantities correspond to some kind of weighted average of the double-layer density over the surface of the particles. The averaging process smooths out the variations of the density across the surface, and as a result, the extracted physical quantities converge much faster: two orders of magnitude for forces and stresslets, and one order of magnitude faster for torques. Therefore, for practical applications it is perhaps sufficient to use error measures based on one of these physical quantities, but here we are interested in evaluating the efficiency of the algorithms and use the absolute error (Eq. 32) as the convergence criterion.

The convergence of the spectral communications scheduling strategy, compared to that of the regular point Jacobi algorithm is shown in Figure 7. We observe that not only does the scheduled algorithm converge faster for any given tolerance (error), but also that its rate of convergence is higher. Considering that the implementation of the schedule $\boldsymbol{N}_{SCS}$ reduces the communications between the processors by approximately 64%, these results are very promising.

Although the schedule used was supposedly deterministic, the asynchronous communications between the processors introduced a degree of randomness in the iteration procedure. Using the information from several runs, we constructed a 99% confidence region for the convergence path (also shown in Figure 7), and as expected, the width of the confidence region is relatively small and essentially constant. Therefore, for practical purposes, this scheduling strategy can be considered deterministic.

The performance of the stochastic schedule $\boldsymbol{P}_{SCS}$, compared to its deterministic counterpart ($\boldsymbol{N}_{SCS}$) and to the regular point Jacobi algorithm, is shown in Figure 8. The 99% confidence region for the convergence path suggests that this algorithm will almost always converge faster than the regular point Jacobi, but that the mean rate of convergence is only slightly higher. We also observe that the deterministic algorithm has a higher convergence rate and, most of the time, runs faster than the stochastic version.

The confidence region shown in Figure 8 reflects the behavior

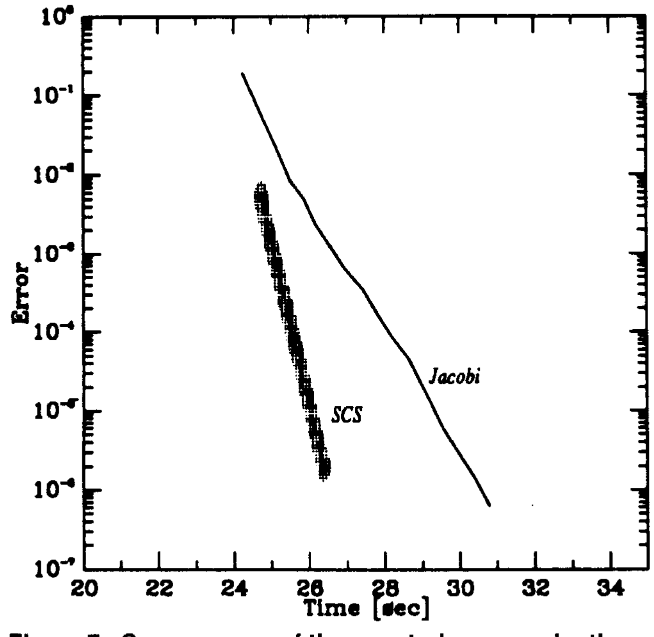

Figure 7. Convergence of the spectral communications schedule, compared to the regular point Jacobi algorithm.

The shaded region corresponds to a 99% confidence region for the convergence path.

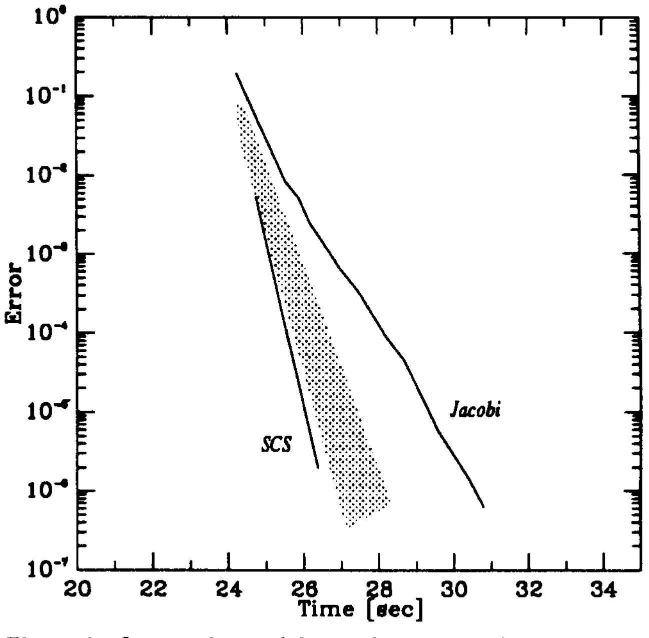

Figure 9. Comparison of the performance of the spectral communications schedule, to a collection of random perturbations of that schedule.

Shaded region enclosed all resulting convergence paths.

of the variance of the solution vector as a function of time. As expected, the confidence region shows an initial divergent trend; however, the algorithm did not iterate long enough to show the expected long-term contraction. It is important to note that if the rate of expansion of the confidence region is slow enough, then the stochastic algorithm can still give a good approximation to the solution of the problem, even if the number of iterations is below the critical number k^*. The results indicate that stochastic scheduling provides a viable solution to the scheduling problem.

A stochastic schedule can be viewed as a sequence of deterministic schedules, which on the average correspond to the spectral communications schedule. However, due to the influence that each schedule has on the iteration of the next one, the mean convergence path will not coincide with that of the deterministic schedule. To illustrate this point we constructed several deterministic schedules $\{N\}$ that corresponded to random disturbances of the spectral communications schedule N_{SCS}.

The resulting convergence paths are enclosed in the gray region in Figure 9, and the similarities with the stochastic results are evident.

Significant differences observed in the behavior of the stochastic and deterministic schedules show how critical is the choice of a good schedule. From Figure 9 it can be argued that the spectral communications schedule is the best schedule for this problem: a collection of random disturbances of the schedule produced convergence paths to the right (slower) of the spectral communications schedule. To show that this might not be just a consequence of a local optimum, we constructed a sequence of totally random schedules with $1 \leq n_{ij} \leq 7$ for $i \neq j$ and $n_{ii} = 1$. The resulting convergence paths are shown in Figure 10. These results do not prove that the spectral communications scheduling strategy generates an optimal schedule, but it does show that it produces a very *competitive* candidate.

We believe that for large-scale systems, screening effects will become important and that a scheduling strategy based on interparticle distances might give excessively conservative

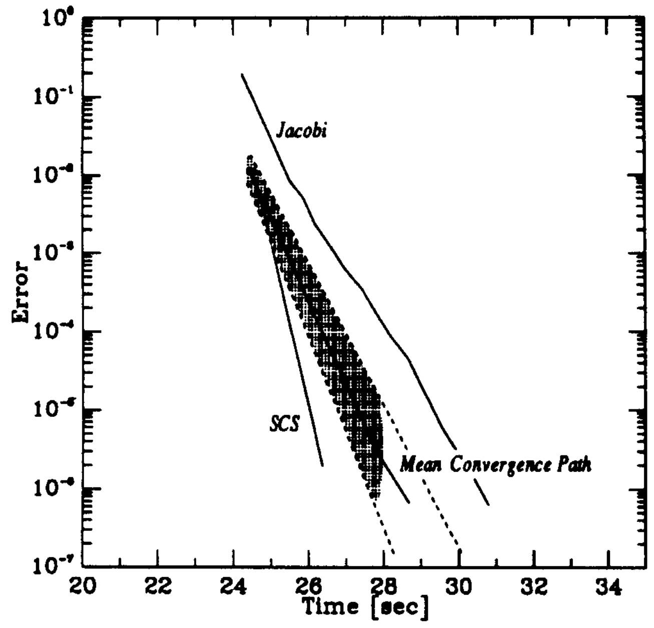

Figure 8. Convergence of the stochastic spectral communications schedule, compared to the deterministic spectral communications schedule, and the regular point Jacobi algorithm.

The shaded region corresponds to a 99% confidence region for the convergence path.

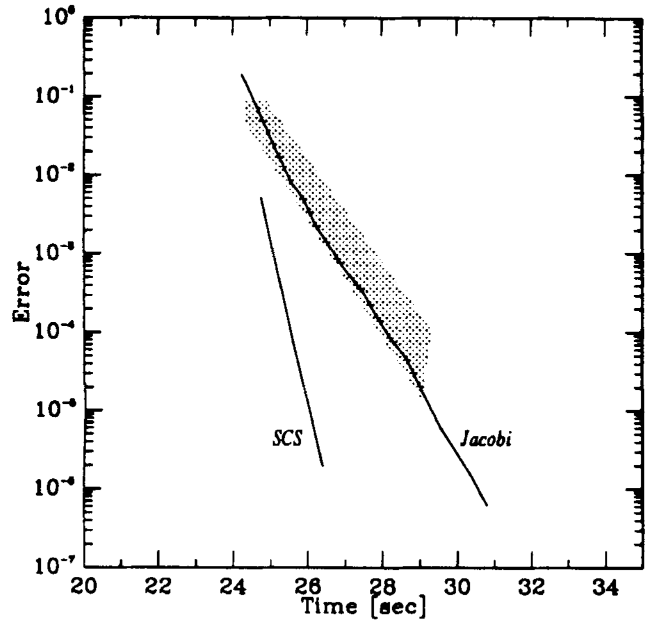

Figure 10. Comparison of the performance of the spectral communications schedule, to a collection of random schedules.

Shaded region enclosed all resulting convergence paths.

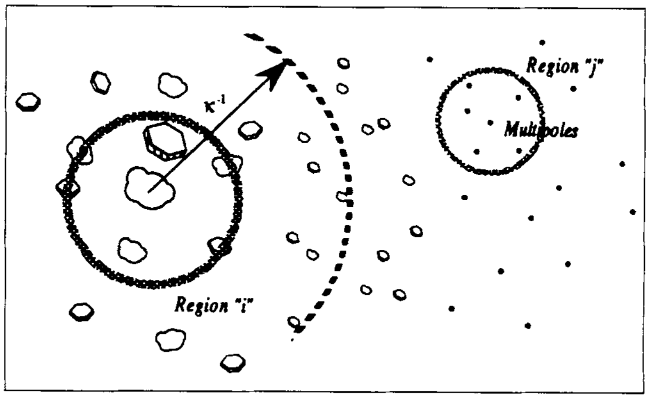

Figure 11. Interactions between regions "*i*" and "*j*": from a complete description of and frequent updates from near neighbors to compactified information sent from distant and screened multipoles.

schedules. Furthermore, the cost of establishing a good schedule in a dynamic manner may be prohibitively expensive. A simple rule based on the *spectral communications schedule*, in either deterministic or stochastic mode, with a suitable modification to account for screening, could provide a good starting point in the search for robust schedules.

Scalable Algorithms for Massively Parallel Computers

The results presented here lead quite naturally to *speculations* concerning the next phase of research—communication issues in larger problems on computers with many more processors. In Figure 11 we consider a typical situation in a suspension. Consider the set of particles in region *i* (this region may range from one to many particles, depending on the amount of local memory available per processor and the complexity of the particle geometry). The boundary integral methods capture the full effect of particle geometry on interactions between neighboring particles. On the other hand, all distant particles (say in region *j*) appear as point-particles. The boundary integral contributions from region *j* may be approximated by a multipole expansion involving gradients of the Stokes-dipole and moments of the dyad φn. The first term in this expansion appears as:

$$(S\cdot\nabla)\cdot\frac{\mathcal{G}(x-r_j)}{8\pi\mu}$$

where

$$S=-2\mu\oint_{S_j}(\varphi n+n\varphi)dS$$

is the all-important stresslet used in calculating the rheological properties of the suspension (Batchelor, 1970).

At this point, we have a situation that is analogous to the Greengard and Rokhlin (1987) algorithm for molecular dynamics. During the construction of the equations associated with region *i*, the multipole expansion is used in place of the full matrix of interactions with boundary elements in region *j*. What is the implication for interprocessor communication? A decrease in the length of the message, which in the architecture of the future, most probably implies fewer packets and thus a reduction in the traffic level. The large system of equations is governed by a matrix with the effective structure as given in Figure 12.

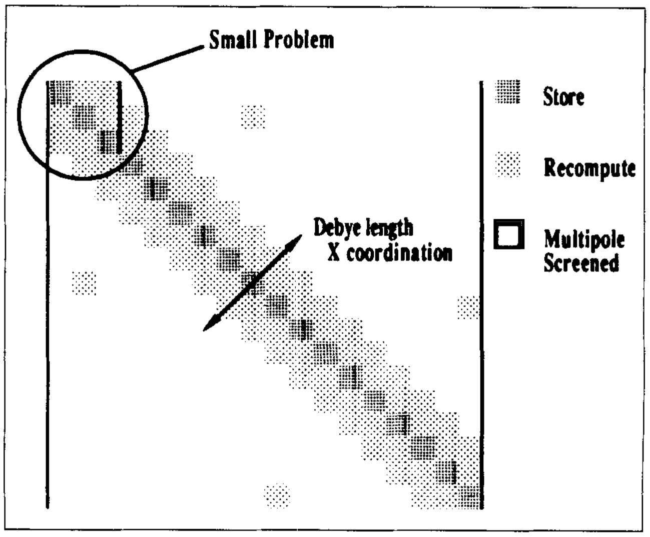

Figure 12. Interactions between regions "*i*" and "*j*": from a complete description of and frequent updates from near neighbors to compactified information sent from distant and screened multipoles.

Computing environment of the future

As we move to more concentrated systems, other physical effects such as renormalization theory and *Debye-Brinkman screening* enter into the picture (Kim and Russel, 1985). The cloud of particles in the medium between regions i and j effectively occlude direct interactions. Screening reduces the *frequency* of interprocessor communication to below the already low levels dictated by the spectral communication scheduling strategy. Using ideas from effective medium theory, we are naturally led to communication frequencies (for both deterministic or stochastic communication strategies) that scale as:

$$\frac{\exp\{-\kappa(c)r_{ij}\}}{r_{ij}^{\nu}}$$

where the screening length κ^{-1} would shrink with increasing particle volume fraction c. The exact nature of κ and ν would depend on the physics of the problem: fixed beds, force-free suspensions, the nature of interparticle colloidal forces, and so on.

The ramification for memory requirements are also significant. Boundary integral equations result in dense linear systems of equations. As the size of the problem is increased, one eventually exceeds even the TeraBytes of memory projected for the next generation of parallel computers. With our iterative approach, we can continue beyond this limit by simply recalculating the matrix (we store the diagonal blocks; the ij off-diagonal block is recalculated using processor j, if and then just before, the message containing the solution vector has to be sent from j to i). Screening introduces an effective bandwidth that scales with κ^{-1}, so that the time spent recalculating the matrix scales as N and not N^2. Figure 13 previews how these ideas may be implemented on a massively parallel computer of the future. We envision it to have the order of 16,000 superprocessors with megabytes of local memory per processor and an interconnect scheme with a finite number of "near-neighbors." Furthermore, the interconnection network, if it is not overloaded, may be able to deliver message transit time that is nearly identical to the distance between processors by using wormhole routing (Dally and Seitz, 1987; Adve and Vernon, 1991). Consequently, the design of the simulator focuses on reducing communication traffic density given that each processor will be responsible for a set of particles in a region of the suspension.

Locally stored information

On processor i responsible for region i, the local memory must contain critical information that is frequently accessed, such as the boundary element geometrical information for region i, information on the parameters for the scheduling rules such as $\kappa(c)$ and ν, the data needed to construct the range completer, and of course, the double-layer density φ in the surfaces of region i. Finally, the N position vectors that pinpoint the location of the centers of all particles in the simulation must also be stored locally. After each time step, the new positions are broadcast from a central supervisor.

Interprocessor communication: which message and when

In the communication phase of the computation, processor

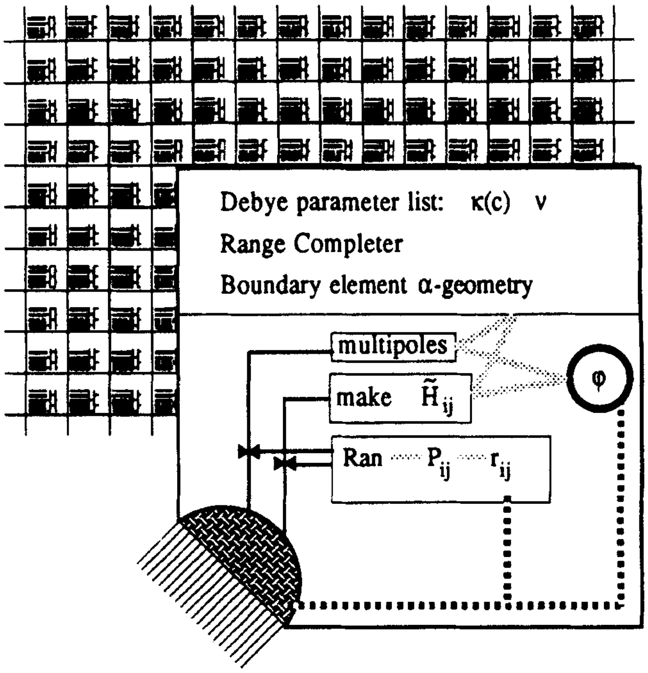

Figure 13. Functional capabilities distributed to each processing node on a massively parallel computer.

j sends out its (local) knowledge of $\varphi(\xi)$ on surfaces in its jurisdiction. When does this communication event take place? In the deterministic schedule, a message is sent to i depending on some prearranged rule for $i-j$ interactions. In the stochastic version, there is a probability p_{ij} in which the message will be sent out. In both versions, the frequency of messages sent decreases with increasing separation r_{ij}—we may even set it to zero beyond a cutoff distance. Of course, this violates one of the assumptions of the asynchronous convergence theorem, and therefore the convergence is not guaranteed any more. We believe on physical grounds that the system will still converge to the desired solution as can be checked with different scheduling parameters to insure that the results are reliable.

If a message is to be sent (to i), what will be sent? If region i is a neighboring region, $\tilde{\mathcal{H}}_{ij}$ must be computed and i must receive $\tilde{\mathcal{H}}_{ij}\cdot\varphi_j$ as well as the boundary element geometry of region j. Conversely, the boundary element geometry of region i is known to j, thanks to a previous communication. On the other hand, if region i is far away, the time-consuming calculation of $\tilde{\mathcal{H}}_{ij}$ is bypassed; only the multipole moments:

$$S=-2\mu\oint_S(\varphi n+n\varphi)dS,\ldots,$$

and the particle center(s) in region j, $\{r_j\}$ are sent. An integrated property over surfaces in region j, suitably accumulated all along inside processor j, is all that processor i needs to update the φ_i in its jurisdiction.

Laplace equation

The computational methods presented here are directly ap-

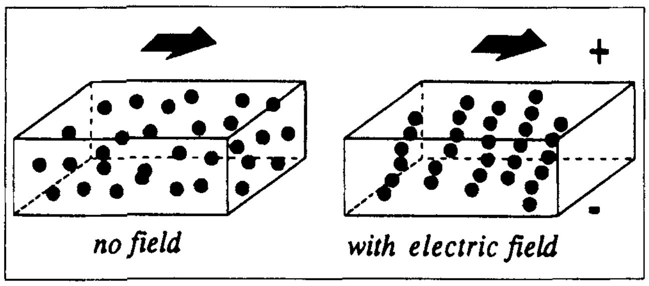

Figure 14. ER fluids: changes in microstructure induced by an applied electric field.

plicable to the solution of the three-dimensional Laplace equations. Indeed, much of the mathematical foundations for the Laplace equation (the existence of a double-layer representation and the analysis of the spectrum of the double-layer operator) predate the corresponding development for the Stokes equations (see, for example, the exercise on potential theory in Friedman, 1982). Furthermore, Wielandt deflations are more straightforward for the electric double-layer operator.

A simultaneous treatment of cooperative electrostatic and hydrodynamic interactions between many particles of arbitrary shape, especially in bounded domains, is of importance in the modeling of the class of materials known as electro-rheological fluids (ER). ER fluids consist of a dispersion of colloidal particles in a (usually) nonaqueous fluid (Gast and Zukoski, 1989). The normal suspension rheology behavior is greatly altered by the application of an electric field. Electric dipoles are induced in the particles; the resulting dipole–dipole interactions lead to structure formation, as shown in Figure 14, as well as a dramatic increase in suspension viscosity and an apparent yield stress. The evolution of this suspension microstructure is an N-particle-cooperative phenomena in which both near-near and near-far interactions play a role, requiring just the approach discussed here. In a future work, we will consider mapping strategies for ER simulations on scalable architectures.

Conclusions

The main points in this work are as follows:

• Computer simulations play an important role in bridging microstructure to macroscopic phenomena, and thus our efforts to bring fundamental science to bear on materials processing. These simulations are compute-intensive and require computing performance that greatly exceed the capabilities of present-day computers.

• The future high-performance computing environment is envisioned as a massively parallel computer consisting of many processors; each processor performs at a level comparable to current supercomputers.

• We expect communication bottlenecks on massively parallel computers to be a major obstacle in the scale-up of computing performance. The key objective of the present study was to devise strategies for minimizing interprocessor communications, thus facilitating scale-up to large-scale simulations.

• The CDL-BIEM algorithm for particle mobilities in a viscous fluid involves a boundary integral representation that converts the governing three-dimensional PDE to an equivalent two-dimensional integral equation. The relevant integral operator is a contraction mapping, and fixed-point iteration schemes are possible. The iterative solution approach naturally introduces an "access-to-memory" hierarchy that separates local and distant communications.

• The effect of local iterations was studied in detail on the two-particle problem. Successive local iterations effectively collapse the spectral radius as seen by the global iterations and thus reduce the demand for global communications.

• The two-particle results can be used in a pairwise fashion to develop an efficient communication strategy for N-particle problems, where N is of the same order as the number of processors in current coarse-grained parallel computers, such as the Intel iPSC/860. Both deterministic and stochastic communication schedules gave encouraging results.

• The techniques and concepts discussed here work just as well to the three-dimensional Laplace equation and thus apply to mathematical models of electrostatics, heat conduction and Darcy flow (flow through porous media). A simultaneous solution of the Stokes and Laplace equation, for example, to simulate the behavior of ER fluids, is also feasible.

Acknowledgment

We acknowledge helpful discussions with Prof. Mary Vernon and Mr. Vikram Adve of the UW-Madison Computer Sciences Dept. This material is based on projects supported by NSF grant CTS-9015377, ONR contract N00014-89J-3001, and the ACS Petroleum Research Fund. Computational facilities were provided by the Center for Parallel Computation in Engineering, UW-Madison, and the Intel Corporation University Partners Program.

Notation

c =	concentration (volume fraction)
$\boldsymbol{D}$ =	distance matrix
E =	rate of energy dissipation
$\boldsymbol{e}_i$ =	unit Cartesian coordinate vector
$\boldsymbol{F}_\alpha^e$ =	external force on particle α
f =	fraction sequential content in an algorithm
$\mathcal{G}$ =	Oseen tensor (Green's dyadic for Stokes equation)
$\mathcal{H}$ =	Wielandt-deflated double-layer operator
I_i =	moment of inertia about ith coordinate axis
$\mathbf{K}$ =	kernel of the double-layer operator
$\mathcal{K}$ =	double-layer operator
M =	matrix
$\boldsymbol{M}$ =	delay matrix
$\boldsymbol{N}$ =	deterministic communication schedule matrix
N_p =	number of particles
$\boldsymbol{n}$ =	unit surface normal
$\boldsymbol{P}$ =	stochastic communication schedule matrix
$\mathcal{P}$ =	projection operator
p =	number of processors
p =	pressure
p_{ij} =	assigned probability that processor j sends update to i
r_m =	ratio of spectral radii: m-delay/no-delay
S_p =	speedup gained by using p processors
S_α =	surface of particle α
$\boldsymbol{S}$ =	stresslet
T =	computational time
T^i =	set of times at which ith variable is updated
$\boldsymbol{T}_\alpha^e$ =	external torque on particle α
$\boldsymbol{U}$ =	particle translational velocity
$\boldsymbol{v}$ =	velocity
x =	unknown vector in linear system of equations
$\boldsymbol{x}$ =	position vector
$\boldsymbol{x}_\alpha$ =	position of particle center

Greek letters

α	=	particle label
δ	=	gap between spheres
δ_{ij}	=	Kronecker delta
κ	=	inverse of Debye-Brinkman screening length
λ	=	eigenvalue of double-layer operator
μ	=	viscosity
$\rho(\ )$	=	spectral radius of a matrix
Σ	=	stress field of $\mathcal{G}/8\pi\mu$
σ	=	stress tensor
τ_j^i	=	time jth variable is available at processor updating the ith
φ	=	double-layer density
ω	=	rotational velocity

Subscripts

SCS	=	spectral communication scheduling

Literature Cited

Adve, V. S., and M. K. Vernon, "Performance Analysis of Multiprocessor Mesh Interconnection Networks with Wormhole Routing," *IEEE Trans. Parallel and Distributed Systems*, submitted (1991).

Andre, F., D. Herman, and J. P. Verjus, *Synchronization of Parallel Programs*, MIT Press, Cambridge (1985).

Batchelor, G. K., "The Stress System in a Suspension of Force-free Particles," *J. Fluid Mech.*, **41**, 545 (1970).

Bertsekas, D. P., "Distributed Asynchronous Computation of Fixed Points," *Math. Prog.*, **27**, 107 (1983).

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

Brawer, S., *Introduction to Parallel Programming*, Academic Press, New York (1989).

Chazan, D., and W. L. Miranker, "Chaotic Relaxation," *Lin. Algeb. and Appl.*, **2**, 199 (1969).

Dally, W. J., and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers*, **C-36**, 547 (1987).

Deng, Y., J. Glimm, and D. H. Sharp, "Perspectives on Parallel Computing," *DAEDALUS* (*J. Amer. Acad. Arts and Sci.*), **Winter 92**, 31 (1992).

Dongarra, J. J., "Performance of Various Computers Using Standard Linear Equations Software," *Supercomputing Rev.* **3**, 49 (1990).

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

Gast, A. P., and C. F. Zukoski, "Electrorheological Fluids as Colloidal Suspensions," *Adv. Coll. Sci. and Interface Sci.*, **30**, 153 (1989).

Greengard, L., and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *J. Comp. Phys.*, **73**, 325 (1987).

Karrila, S. J., 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. Commun.*, **82**, 123 (1989).

Karrila, S. J., Y. O. Fuentes, and S. Kim, "Parallel Computational Strategies for Hydrodynamic Interactions between Rigid Particles of Arbitrary Shape in a Viscous Fluid," *J. Rheol.*, **33**, 913 (1989).

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

Kim, S., and W. B. Russel, "Modeling of Porous Media by Renormalization of the Stokes Equations," *J. Fluid Mech.*, **154**, 269 (1985).

Manber, U., *Introduction to Algorithms*, Addison-Wesley, Reading, MA (1989).

Odqvist, F. K. G., "On the Boundary Value Problems in Hydrodynamics of Viscous Fluids," *Math. Z.*, **32**, 329 (1930).

Pakdel, P., and S. Kim, "Mobility and Stresslet Functions of Particles with Rough Surfaces in Viscous Fluids: A Numerical Study," *J. Rheol.*, **35**, 797 (1991).

Phan-Thien, N., D. Tullock, and S. Kim, "Completed Double Layer in Half-Space: a Boundary Element Method," *Computational Mechanics*, **9**, 121 (1992).

Power, H., and G. Miranda, "Second Kind Integral Equation Formulation of Stokes' Flows past a Particle of Arbitrary Shape," *SIAM J. Appl. Math.*, **47**, 689 (1987).

Quinn, M. J., and N. Deo, "Parallel Graph Algorithms," *Comput. Surv.*, **16**, 338 (1984).

Ramkrishna, D., and N. R. Amundson, *Linear Operator Methods in Chemical Engineering*, Prentice-Hall, Englewood Cliffs, NJ (1985).

Stone, H. S., *High-Performance Computer Architectures*, Addison-Wesley, Reading, MA (1987).

Manuscript received Oct. 15, 1991, and revision received Apr. 10, 1992.