Engineering with Computers (2001) 17: 1–15

Engineering with Computers

Discrete Element Modelling on a Cluster of Workstations

J. W. Baugh Jr. and R. K. S. Konduri

Department of Civil Engineering, North Carolina State University, Raleigh, NC, USA

Abstract. *We describe a distributed computing system for discrete element modelling that has been designed for loosely-coupled networks of workstations. The implementation is based on DM², a state-of-the-art discrete element modelling technique for simulating the behaviour of energetic materials and modelling shock compaction phenomena. The underlying computational approach is derived from particle methods, where short-range interactions, both mechanical and thermochemical, determine individual particle movement and state. Using spatial decomposition, a client-server software architecture distributes the computations and, at the language level, Berkeley sockets enable communication between conventional Unix processes on workstations connected by an Ethernet. We evaluate the performance of the system in terms of overall execution time and efficiency, and develop a simple model of computational and communication costs that enables us to predict its performance in other contexts. We conclude that distributed implementations of short-range particle methods can be very effective, even on non-dedicated communication networks.*

Keywords. Distributed computing; Particle methods; Short-range interactions

1. Introduction

A new computational approach is emerging for simulating the mechanical and thermochemical behaviour of materials, with such diverse markets and applications as pharmaceuticals, bulk chemicals, oil exploration and environmental hazard mitigation. Instead of relying on continuum models, particle methods – as they are sometimes called – model the interaction of atoms, molecules or grains, producing macroscopic behaviour that is not easily captured, if at all, by conventional means. Related but distinct approaches are those with spatially discrete domains, e.g. cellular automata methods such as lattice-Boltzman, in which particles reside only at discrete lattice points.

Depending on the application, particle systems may be simulated at various scales of time and space. For example, in *molecular dynamics* applications, solids and liquids modelled at the level of atoms and molecules can be simulated for picoseconds of actual time [1]. At a coarser resolution, *mesoscale* models of individual grains, crystals or particles can be studied at the nanosecond scale, allowing corroboration with experimental results [2]. Problems at still a larger scale have been simulated using *Discrete Element Methods* (DEM), in which test specimens are beginning to approach the size of those routinely used in engineering experimentation [3]. At the latter scales, particle methods will have an increasingly important role, particularly for phenomena that experience mixed regimes of behaviour, e.g. soils that transition from solid deformation to granular flow. In geomechanics alone, they may substantially improve our ability to predict the occurrence of catastrophic landslides, or to determine the effect of seismic events on building structures.

While enabling the simulation of diverse phenomena, particle methods have enormous computational requirements, which are determined by the number of particles in the simulation and the simulation time. Developments in high performance computing, however, have led to codes capable of simulating realistic systems with hundreds of millions of particles. These research efforts have largely focused on the development of algorithms for tightly-coupled parallel machines and supercomputers.

This paper describes the development and performance of a discrete element code that runs well on a loosely-coupled cluster of workstations. The implementation is based on DM², a discrete element technique developed at NCSU and Los Alamos National Laboratory for modelling energetic materials

Correspondence and offprint requests to: Dr J. W. Baugh Jr, Department of Civil Engineering, North Carolina State University, Raleigh, NC 27695, USA

at the mesoscale. Also described is a performance model that allows us to predict the timing behaviour of the code in different contexts, e.g. on high-end networking hardware, or with a different communication strategy. We begin with some background and related work before describing our programming environment, sequential and distributed implementations, and performance model.

2. Background and Related Work

Research in particle methods has a long history, with hundreds of references published by 1973 [4]. Work in molecular dynamics was reported as early as 1959 [5], and engineering applications of discrete element methods appeared as early as 1971 [6]. While early studies in DEM typically involved small systems and focused on granular materials [7,8], more recent efforts with improved algorithms and modern computational resources have yielded larger simulations and diverse applications, including analysis of hydraulic problems [9], viscoelastic behaviour of hot mix asphalt [10], and microscopic modelling of soils [11].

Although the focus of this paper is on DEM, as opposed to particle methods in general, many of the underlying concepts, algorithms and representations are similar. Here we introduce some of those computational and parallelisation issues, and then describe the modelling capabilities of DM^2.

2.1. Computational Approach

Given an ensemble of n interacting particles, Newton's equations of motion may be solved to determine particle trajectories, i.e.

$$m_i \frac{dv_i}{dt} = \sum_j^n F(r_i, r_j) \quad \frac{dr_i}{dt} = v_i \tag{1}$$

where m_i, r_i and v_i are the mass, position and velocity of particle i, respectively, and $F(r_i, r_j)$ is the pairwise force between particles i and j, the latter of which may be generalised to n-body interactions; a similar relation holds for rotational motion. In molecular dynamics simulations, interactions may be governed by interatomic forces, such as the well-known Lennard-Jones 12–6 potential, originally proposed for liquid argon. For discrete element modelling, interacting forces may be represented by mechanical elements, e.g. damped springs in the normal direction, springs in series with frictional sliders, etc.

After an initialisation phase, the basic algorithm consists of a repetition of the following steps:

(1) determine forces, e.g. interaction computations, contact detection;
(2) integrate equations of motion to determine updated positions, etc.;
(3) take measurements, e.g. aggregate properties such as strain;
(4) increment time step.

Of these steps, the most computationally demanding is the first. For problems involving *long-range* forces, several approximate algorithms are available for reducing the $O(n^2)$ computations required to determine interaction forces, including hierarchical methods, which are $O(n \log n)$ [12], and fast multiple methods, which are $O(n)$ [13]. When particle interaction is governed only by *short-range* forces, as is the case for the mesoscale and discrete element models considered in this study, the necessary force computations can be reduced with two techniques that can be used alone or in combination:

- *Neighbour lists* [14]. When a particle is known to interact only with particles within a radius R_c, a neighbour list can be constructed for it that includes all particles within a radius $R_c + \Delta R$, where ΔR is small and empirically determined. Because particle movement is small between time steps, this list can be used to determine neighbors within R_c for several iterations, perhaps 10 or 20, at which time the list is recomputed.
- *Cell subdivision* [15]. By subdividing the simulation region into square- or cube-shaped cells, as shown in Fig. 1, every particle can be assigned to a unique cell based on its position. If the edge

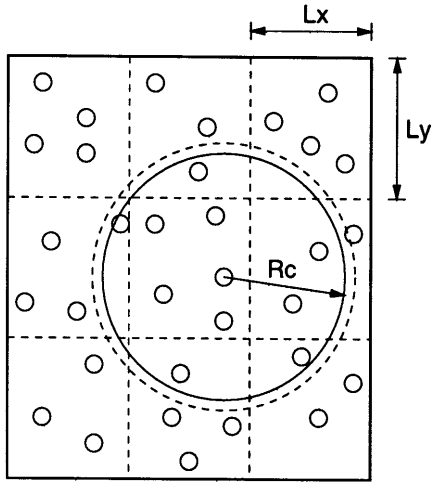

Fig. 1. Cell subdivision.

lengths of the cells, L_x and L_y, exceed R_c, a given particle will interact with at most the particles in its own cell or its surrounding cells. Such an approach reduces the number of interaction calculations to $O(n)$.

2.2. Parallelisation

Because of their computational requirements and inherent parallelism, particle methods have received considerable attention with respect to their implementation on high performance computers [16]. Implementations and studies have been performed on various architectures, including SIMD machines such as the CM-5 [17] and MIMD machines with dozens to thousands of processors [1,18].

Parallelisation of the steps described above typically centres on the first, since the others are either trivial or trivially parallelised. This step, the determination of interaction forces, is often complicated by the need for processor coordination, since some of the necessary data may appear on other processors. Plimpton [1] describes three approaches for its parallelisation:

- *Particle decomposition* (or *atom* decomposition). By assigning particles to processors, a subset of force computations and position updates is performed by each. Because a particle's neighbour may reside on another processor, an exchange of data is necessary, although replication can be used to perform these exchanges *en masse* at successive time steps. While load balancing is not a problem, communication is $O(n)$, limiting the number of processors that can be effectively employed.
- *Force decomposition*. If particle decomposition is viewed as a row-wise decomposition of the $n\times n$ matrix of interacting forces between particles i and j, force decomposition is a partitioning of that matrix by blocks. Such an approach maintains the load-balancing features of the prior approach, while reducing communication cost to $O(n/\sqrt{N})$, where N is the number of processors.
- *Spatial decomposition*. Space is subdivided across processors so that each performs computations on the particles in its own subregion. Because of particle movement at each time step, some particles may move from one processor to another. For some problems, a dynamic load-balancing scheme may be required to maintain good performance and the optimal $O(n/N)$ communication cost of the algorithm.

While general guidelines for selecting a decomposition approach are beginning to emerge for some architectures, variations in processor number and characteristics, network topology, communication latencies and bandwidth, memory hierarchies and the problem domain characteristics themselves frustrate attempts to make good, *a priori* design commitments. In addition, the complexity of the structural and bookkeeping operations required by an approach and its innumerable variations make after-the-fact changes difficult; as a result, it is rarely easy to show that hardware is being used effectively.

2.3. Discrete Element Modelling at NCSU

Prior research at NCSU and Los Alamos National Laboratory led by Horie [2,19] resulted in DM^2 (Discrete Meso-Dynamic Modelling), a discrete element code featuring

(1) a multi-physics capability to describe both mechanical and thermochemical processes including phase change and chemical reaction at the meso-level;
(2) a multi-element representation of particles and grains to deal with complex morphology and large inelastic deformation, and
(3) a connectivity algorithm to deal with flaws, voids, interfacial properties, and other microstructural heterogeneities.

DM^2 has been successfully used to model a variety of materials subjected to dynamic or quasi-static loading at the meso-level. Examples are inert shock response of HMX powder, chemical reaction in a shear band, granular shear flow and shock compression of polycrystalline copper.

Capabilities of DM^2

Mechanical interactions consist of a radial force based on the central potential, a central damping force, dry friction, a tangential viscous force and shear resistance. The central potential is represented by such functions as the Lennard-Jones potential, the Morse potential and a quadratic potential, depending on the material under investigation. Material parameters are evaluated by using Hugoniot data for compression. In tension, the function is supplemented by a temperature dependent yielding function. Shear resistance is described by an elastic-perfectly plastic model. The central damping force is a linear function of the radial relative velocity, and the damping coefficient a constant or function of pressure and temperature. Dry friction is based

on Coulomb's law. Stress and strain in DM^2 are based on a single meso-element and its interactions with neighbouring elements. Meso-level stress and strain are defined at the centroid of an element, the former being based on momentum balance, and the latter on an average Eulerian strain using incremental displacements relative to neighbouring elements.

Thermochemical states are defined by several parameters including temperature, pressure, composition and phase. The total temperature change of an element consists of the change due to heat conduction and the energy dissipated by inelastic deformation such as viscous damping and friction. Phase transitions are currently limited to condensed phase. Presently, no convecting gas phase is considered. The transitions are assumed to be instantaneous, and the threshold conditions are specified by a function of pressure and temperature.

Thermochemical interactions are assumed to occur only between contacting elements, and consist of heat conduction and chemical reaction. The former is based on Fourier's law. The reaction model can deal with both stoichiometric and non-stoichiometric reactions. The current model conserves the number of particles after the reaction.

3. Computing Environment

A distributed computing system consists of a collection of autonomous computers connected by an interconnection network, and equipped with software that enables them to coordinate activities and share resources. Laptops, personal computers, high-end engineering workstations and symmetric multiprocessors can all contribute in a heterogeneous, distributed computing environment. The growing power of individual processors, coupled with the availability of high-speed networks, is quickly making distributed computing a cost-effective alternative to supercomputers and special-purpose parallel computers for solving large, computationally intense problems.

Anderson et al. [20] note that the case for workstation clusters is stronger than ever, given (1) the growing availability of switched networks that scale well with the number of processors, (2) the extraordinary performance of modern workstations, and (3) the I/O bottleneck that makes 'memory over the network' less costly than disk I/O. Our own studies since the late 1980s have shown that distributed solutions on non-dedicated, heterogeneous hardware are a practical approach in such diverse application areas as finite element analysis [21,22], vehicle routing and scheduling [23], and air quality optimisation and management [24].

3.1. Software Architecture

The software architecture adopted in this study is based on the client-server model, which is commonly used in network applications, such as those for file transfers, remote logins and front ends for shared or spooled resources. The basic idea in the client-server model is to provide specialised servers that are capable of performing parts of a larger computational process together with a client that uses them to solve the more general problem. Thus, an efficient use of resources is achieved by hierarchically decomposing problems into subproblems that are distributed and solved concurrently.

Unlike that of peer-to-peer architectures, the relationship between processes in the client-server model is an asymmetric one: initiating interprocess communication requires that processes know which role they are to play, whether client or server, so that they perform the appropriate actions. A *server* is a process that, once invoked, waits to be contacted by a client requesting service. Only when a client requests a connection does a server become active. After the server processes the request, it goes back to sleep, waiting for another request to arrive. Servers that hand off requests by forking new threads to service them are termed *concurrent*, since they can handle new requests before completing prior ones. Servers that process requests one-at-a-time are termed *iterative*, and are typically more appropriate for high-performance applications.

A *client* is a process that, when invoked by the user, decomposes the problem into parts and assigns them to servers by first establishing a connection and then sending its request. Data is communicated by passing messages with non-blocking *writes* and blocking *reads*. That is, the sender proceeds after writing without waiting for the receiver to get the message, unlike the receiver which must wait until it is received.

3.2. Program Development

The programs implemented in this study were developed in the C programming language using BSD Sockets (Berkeley Software Distribution) for interprocessor communication. While numerous tools and languages are available for distributed computing (e.g. PVM, MPI, ISIS and Linda), sockets offer both flexibility and efficiency in developing distrib-

uted computing solutions. In addition, sockets are an accepted industry standard that are implemented on a variety of platforms, and are inexpensive for the application in terms of both memory and performance.

For interprocessor communication both client and server issue *socket*() calls, as shown in Fig. 2. A socket is an endpoint for communication, and during creation the communication domain, type, and protocol are specified, e.g. AF_INET for the Internet protocol and SOCK_STREAM for reliable, connection-based streams. On the server side, a local address and port are assigned to a socket using *bind*(), which makes the socket visible. Then, a *listen*() call indicates a willingness to accept incoming connections, and connections are accepted with *accept*(). On the client side, a connection with the other socket is attempted with *connect*(), which specifies its address.

Once a connection is established between the sockets, data can be transmitted by using the same I/O system calls, *read*() and *write*(), that are used for ordinary files. The communication is *full duplex*, meaning that each end can act as a sender or a receiver. Complex structures can be communicated over the network by marshalling and unmarshalling data on the sending and receiving ends, respectively. When no further communication is required, each end of the socket connection is closed by the respective process.

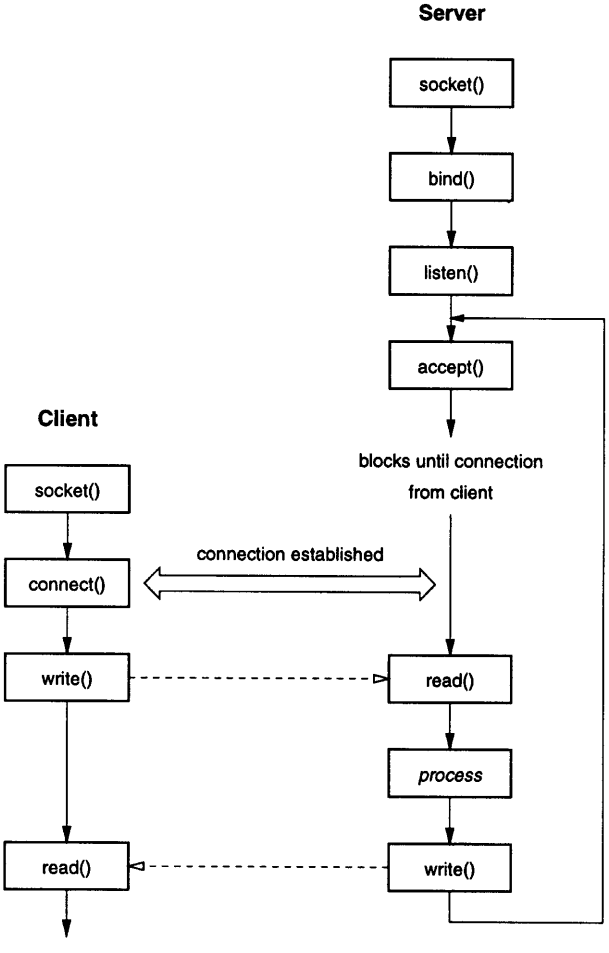

Fig. 2. Socket calls between client and server.

3.3. Computing and Network Hardware

The experiments we report were performed on a cluster of Sun Ultra-10 Workstations with 128 MB of RAM running Solaris 2.5 on a 10BaseT Ethernet (10 Mbps). Ethernet (IEEE 802.3) is a broadcast bus technology: hosts on the network share a single communication link and data must be broadcast to reach any other host. The destination host interface then accepts packets addressed to it and filters the rest.

The unit of transmission on an Ethernet is the *frame*. The minimum size of a frame is 64 bytes and the maximum 1500 bytes. Frames smaller than 64 bytes are padded while those larger than 1500 bytes are fragmented. The transmission of frames is based on a distributed medium access control protocol referred to as CSMA/CD (Carrier Sense Multiple Access with Collision Detection). Using CSMA/CD, each host that has data to transmit listens to see if the network is idle. If it is busy, it waits until it becomes idle and then transmits the data. The host continuously listens to the channel for collisions, and abruptly stops transmission when they occur. It then waits for a random amount of time determined by a binary exponential backoff algorithm before transmitting again.

The communication costs in sending data on an Ethernet vary linearly with the length of message sent. In our environment, experiments show these costs for 2, 4 and 8 servers to be (Fig. 3):

$$T^2_{comm}(x) = 3.6 \times 10^{-6}x + 3.3 \times 10^{-3} \quad (2)$$

$$T^4_{comm}(x) = 7.3 \times 10^{-6}x + 4.1 \times 10^{-3} \quad (3)$$

$$T^8_{comm}(x) = 14.6 \times 10^{-6}x + 25.8 \times 10^{-3} \quad (4)$$

where x is the length of message in bytes and T_{comm} is the communication time per cycle in seconds, a cycle being the round-trip communication of a message from one processor (client) to 2, 4 or 8 processors (servers) and back. Time per cycle is measured by timing two consecutive statements: a *write*() followed by a *read*() for the same length of message.

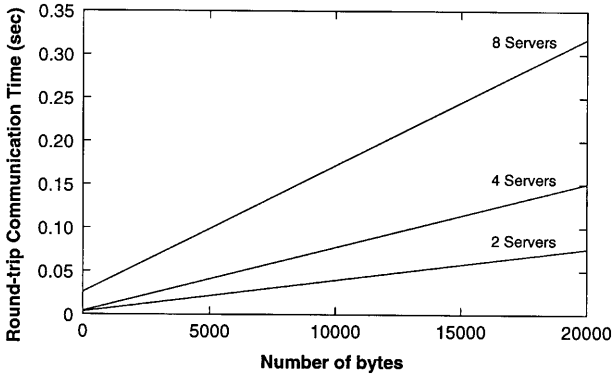

Fig. 3. Communication costs on an Ethernet network.

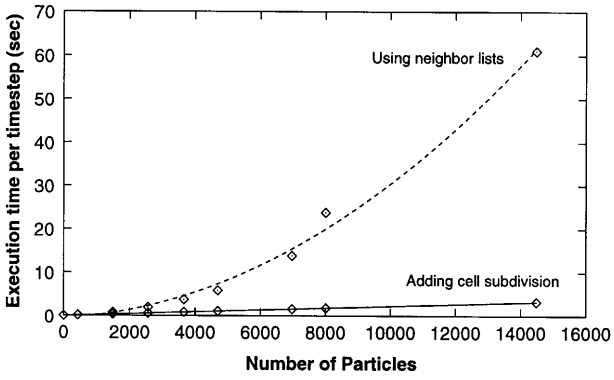

Fig. 4. Performance of neighbour lists and cell subdivision algorithms.

This relationship is a characteristic of the network used, in particular a 9-node subnet during moderate use.

The relative costs of communication to computation in our environment are many orders of magnitude greater than those of switched networks and tightly-coupled parallel machines. As the constant terms show, the fixed charge for communication is significant, especially when compared to the linear terms. The ratio of times for transmitting a word of data and performing a single floating point operation – a common measure of relative communication cost – is on the order of hundreds of thousands; on architectures supporting fine-grained parallelism this ratio is often evenly balanced.

These communication characteristics have important implications for developing distributed systems, which must be reasonably coarse grained, at least on conventional networks. In particular, algorithms should be designed to communicate perhaps larger, but fewer, messages instead of several short messages. Thus, when possible, as much data as practical should be packed into a buffer and sent as a single message.

4. Sequential Implementation

Before describing the distributed implementation, in this section we give a brief overview of enhancements made to the sequential part of the code. The original implementation of DM^2 uses a neighbour list algorithm to avoid recomputing particle neighbours at each time step. In our implementation, cell subdivision is combined with neighbour lists to further improve performance, thereby reducing the interaction calculations to $O(n)$, as shown in Fig. 4.

The algorithm combining cell subdivision and neighbour lists appears in Fig. 5. It begins by

```
algorithm Sequential DM²
var
  t_max, Δt        % maximum simulation time, time step
  dmax, dumax      % current and cumulative maximum displacement
  threshold        % displacement at which neighbors are recalculated
begin
  initialise cells
  generate and assign particles to cells
  determine neighbor lists
  dumax ← 0
  for i ← 0 to t_max/Δt do
      if dumax > threshold then
          update neighbor lists
          dumax ← 0
      endif
      find linked and contact status of particles
      calculate forces and integrate equations of motion
      update positions and velocities of particles
      reassign exiting particles to cells entered
      determine maximum particle displacement, dmax
      dumax ← dumax + dmax
  endfor
end
```

Fig. 5. Implementation combining cell subdivision and neighbour lists.

initialising the cell structure, and in doing so selects cell dimensions so that they exceed $R_c + \Delta R$, guaranteeing that a given particle interacts only with those in the same or in surrounding cells. Particles are then generated and assigned to cells, which are organised in an array-like structure to allow constant access time to neighbouring cells. Given cell dimensions of L_x and L_y, for example, a particle at position (r_x, r_y) is assigned to $cell[i][j]$ such that $i = \lfloor r_x/L_x \rfloor$ and $j = \lfloor r_y/L_y \rfloor$.

Using this cell structure, neighbour lists of particles are determined from adjacent cells once during initialisation, and then intermittently in the iterative part of the algorithm. When this should happen is found by doing a little bookkeeping: at each time step the maximum particle displacement (of all

particles) is recorded as *dmax* and added to a running sum, *dumax*. Unless *dumax* exceeds a *threshold* empirically determined by R_c and the respective particle radii, no new pairs of particles will begin interacting, so the neighbour lists remain valid.

From neighbour lists, pairs of interacting particles are determined at each time step, whether linked, contacting, or both. Contacting particles may be thought of as 'touching', i.e. the distance between the center points of the particles does not exceed the sum of their radii. Linked particles, on the other hand, have a chemical bond that is determined not only by their separation in the current time step, but also by their state, whether linked or unlinked, in the prior time step. That is, particles become linked when their separation is small, and then stay linked unless their separation exceeds a predefined cutoff value.

After finding the interaction status of particles, forces are then calculated, and the equations of motion integrated using the leapfrog method, one of the simplest numerical techniques. In this method, velocities are evaluated midway between the instants at which positions and accelerations are computed. Higher order integration schemes may also be used, but at the cost of additional storage – usually acceleration values retained from previous timestep(s) – and only a minor amount of extra computation. After positions and velocities have been updated, particles leaving a given cell are reassigned to those they enter.

5. Distributed Implementation

Further improvements to performance can be obtained by distributing the computational workload across multiple processors. Obstacles to the efficient utilisation of those processors, however, generally include parallelism and synchronisation overhead, redundancy of computations, inherently sequential sections, load imbalance and communication costs. Of these, the latter two are of particular concern here: because communication costs are likely to be substantial, a spatial decomposition approach is used to partition the computations. Unfortunately, such an approach can lead to imbalances in loads as particles migrate from processor to processor.

- **Load balancing**. The goal of load balancing is simple: assign particles so that processor idle time at synchronisation points is minimised. Using approaches similar to those of finite difference and finite element computations, discrete element models can be decomposed so that processors control either an approximately equal volume of space or number of particles (Fig. 6). For rectangular domains, we use a recursive bisection technique that successively partitions the domain into two equal regions of space with a cut in the direction that minimises interface length, and hence communication costs (since our network topology is a simple bus). While such an approach works well on the problems we have encountered, a static decomposition cannot satisfactorily deal with models that exhibit large particle movements.
- **Communication costs**. Granularity of a distributed algorithm refers to the relative amount of computational work done between processor synchronizations. For short-range interaction problems, the amount of work performed during a single step, and hence between synchronisation points, is much smaller than in the long-range case; this means that, of the various particle methods, discrete element problems can be the most difficult to parallelise efficiently. Mitigating these concerns to an extent is the observation that the computation-to-communication ratio for spatial decomposition grows with n, the number of particles, as $O(\sqrt{n})$.

5.1. Algorithm

Observing that communication costs are a potential concern, we set about to design and implement a distributed system based on spatial decomposition and a client-server architecture. The resulting implementation can be characterised by the top-

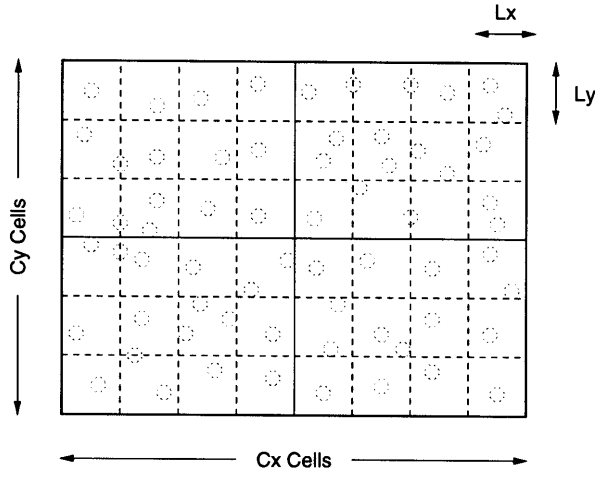

Fig. 6. Domain decomposition.

level flow chart that appears in Fig. 7. In this implementation, servers perform the same tasks as the sequential algorithm on the subdomains assigned to them. Much of the bookkeeping and the overall control, however, resides in the client process.

Before the computation begins, servers bind to a socket and wait for a connection on a well-known port. When the client is invoked, a connection is established with the servers, and then the processes, both client and server, read input data from a shared file that contains basic geometric, material and process control parameters. Based on the configuration parameters in the input file, the client generates a skeletal representation of the system being modelled and determines the total number of cells in the entire domain and the size of each cell. The skeletal domain is then recursively bisected so that subsequent communication is minimised and roughly equal subdomains are assigned to each server. Also

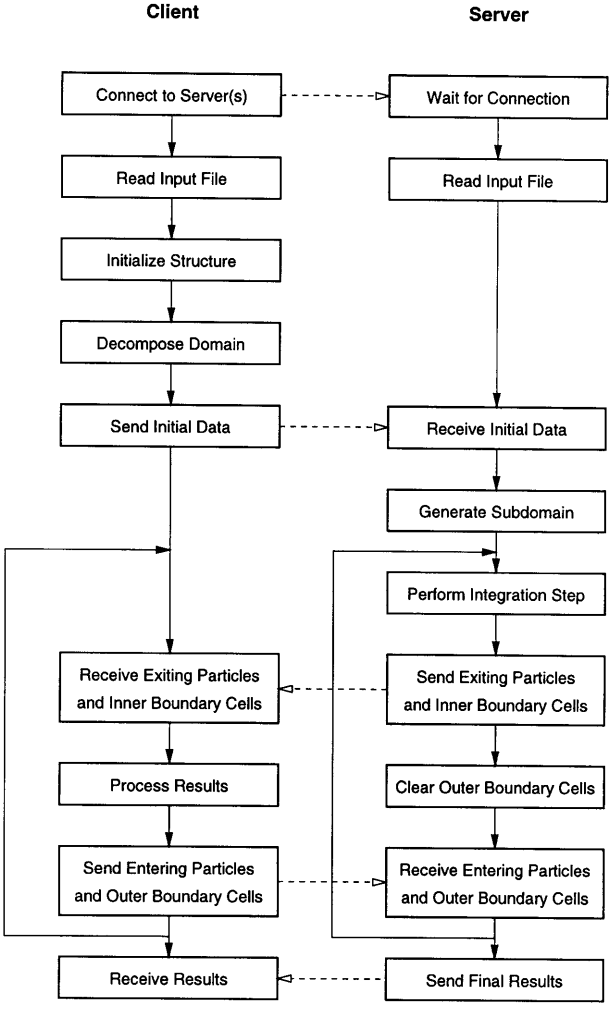

Fig. 7. Top-level flowchart of distributed DM^2 algorithm.

maintained on the client are structures that keep track of server data, including socket and TCP/IP information, coordinates of the inner and outer boundaries, and a buffer containing particles to be migrated to the server.

After reading the input file, servers receive their assignments, which include an eight-element array (four sides and four corners) that specifies which of their boundaries interface with other subdomains. Servers then generate their subdomains by following the basic steps of the sequential algorithm described earlier, i.e. by initialising cells, generating and assigning particles to those cells, and determining neighbour lists. Initial communication costs are reduced by generating particles directly on their respective servers. Servers perform the above steps on cells in their assigned subdomain, making use of the outer boundary cells that are mirrored from other subdomains. The latter cells are 'read-only' in the sense that they are merely used to compute inter-particle forces on particles *within* the assigned subdomain.

Once subdomains are generated, servers perform a single integration step, which includes updating neighbour lists if necessary, finding the linked and contact status of particles, calculating forces and integrating the equations of motion. The updated positions and velocities of particles in the inner boundary cells, which interact with particles in 'neighbouring' processors, are then communicated to the client. Also communicated are the particles exiting the subdomain, which must be migrated to other processors. To reduce the effects of load imbalances, the client issues a *select*() system call to read the data from servers as they become available instead of in a predefined order.

Two aspects of particle migration should be noted here. First, because the linked status of particles depends on their prior state, particles sent to other processors must also include portions of their neighbour lists. Secondly, to facilitate exchange, each particle has associated with it two forms of identification: a global *id* assigned by the client, and a local index in a *particle* array assigned by the server for efficient look up. When neighbour lists are communicated, servers must convert local indices in the neighbour list to global *id*s and back; this is accomplished by maintaining a hash table to map the global *id*s to indices; the reverse mapping is straightforward, since particles 'know' their global *id*s. The size of the hash table is determined at run time based on the number of particles in the subdomain.

While servers clear their outer boundary cells, the

client does some bookkeeping: particles needed by a given server are collected in a buffer as they come in, and are then transmitted. In Fig. 8, for instance, a subset of the inner boundary cells from subdomains 1, 3 and 4 is needed by subdomain 2 to form its (mirrored) outer boundary. Of course, sending particles one-at-a-time is extremely inefficient, as is apparent from the simple communication cost model shown earlier, and so the communication of particles is always buffered by the sender. Also communicated at pre-specified timesteps, though not shown in the flowchart of Fig. 7, are subdomain particle states of interest to the modeller. These states are collected and compiled at the end of the simulation for studying the evolution of the system over time.

6. A Performance Model

Our implementation has been validated with a number of mesoscale models that have been well-studied and themselves corroborated with experimental data [2]. Before presenting the results of one such case, here we describe a simple performance model of our implementation that captures the predominant computational and communication costs. The development of such a model allows us to further validate our implementation, in the sense that it can be shown to respond to changes in model parameters and size, for example, in a predictable manner. In addition, the model allows us to assess the performance of the implementation in other (hypothetical) scenarios, e.g. with faster communication networks such as Gigabit Ethernet or with high-end servers.

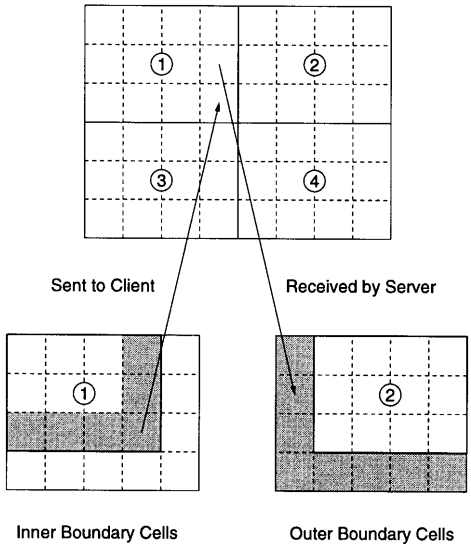

Fig. 8. Cell information communicated between clients and servers.

To construct a performance model for discrete element problems requires that certain simplifications be made. In particular, numerous parameters affecting communication and computation in one way or another cannot be determined without actually performing the simulation. Particles move throughout the simulation causing variations in the amount of computational work within cells, resulting in load imbalances. Likewise, particle migration across processors is only *need-predictable* – the communication is predictable at compile time, but the actual number of particles communicated cannot be known until run time. As a result, we choose to make several approximations that tend to overestimate to a small degree the predicted efficiency.

The performance model developed in this section applies to our sequential and distributed implementations when used to simulate discrete element problems on rectangular domains. It approximates the cost of computation and communication as a function of n, the number of particles in the simulation, and uses empirical results of the setup described earlier to determine constants. The basic parameters are:

s	aspect ratio, width to height, of the domain,
d_x, d_y	centre-to-centre particle spacing in x and y directions,
L_x, L_y	*cell width and height, respectively.*

Using these, cost functions are developed for the sequential implementation, and for the distributed implementation with 2, 4 and 8 servers.

6.1. Sequential Computation Costs

The basic algorithm uses neighbour lists and cell subdivision to reduce the overall computation time, which is a function of the number of cells into which the domain is divided. Within each cell are a number of particles that undergo movement during the simulation.

If we assume a rectangular domain of particles arranged in an orderly fashion, whether vertically or diagonally, its layout can be viewed as an array of rows and columns. The total number of particles, n, therefore, is the product of the number of columns of particles, n_x, and the number of rows, n_y, or

$$n = n_x n_y \tag{5}$$

The width, w, and height, h, of the domain, then,

are approximately the product of the particle separation and number of particles

$$w = d_x n_x, \quad h = d_y n_y \tag{6}$$

Given an aspect ratio s of width to height, it follows that the number of columns and rows of particles is, respectively

$$n_x = \sqrt{s \frac{d_y}{d_x}}\sqrt{n}, \quad n_y = \frac{1}{\sqrt{s \frac{d_y}{d_x}}}\sqrt{n} \tag{7}$$

If cell dimensions L_x and L_y are chosen so that they evenly divide the width and height, the numbers of cells in the x and y directions are

$$c_x = w/L_x, \quad c_y = h/L_y \tag{8}$$

and the total number of cells is

$$c = c_x c_y \tag{9}$$

Given the total number of cells in terms of n, the cost of the overall computation can be determined. We know that the algorithm performs $O(n^2)$ operations for the n particles in a nine-cell window. This computation is performed over c cells, so the computation time is proportional to

$$c\left(9\frac{n}{c}\right)^2 \tag{10}$$

If hardware, compiler, and other system performance attributes are collected into a constant k, then the total sequential computation time is

$$T_{seq} = kc\left(\frac{n}{c}\right)^2 = kn^2/c \tag{11}$$

Or, in terms of the basic model parameters

$$T_{seq} = k\frac{L_x L_y}{d_x d_y} n \tag{12}$$

which, as an aside, shows that cell dimensions should be chosen to be as small as possible. To determine k, a representative discrete element problem with the following constants is chosen: d_x = 0.04 cm, d_y = 0.03464 cm, $L_x = L_y$ = 0.121 cm. Then, from experimentation within the environment described in Section 3, the following is obtained:

$$T_{seq} = 0.3316n \text{ (in ms)} \tag{13}$$

Solving for k yields k = 0.0314, again, where k is some measure of processor and compiler performance.

6.2. Communication Costs

For the distributed implementation, communication costs are proportional to the cumulative length of the subdomain interfaces and the average particle density of the cells along those interfaces. Because of the rectangular structure of our problems, subdomain interfaces are fractions of either rows or columns of cells, or possibly their combination. We can approximate the total number of particles p_x that appear in a row of cells or the number p_y that appear in a column as follows:

$$p_x = c_x\left(\frac{n}{c}\right) = L_y\sqrt{\frac{s}{d_x d_y}}\sqrt{n} \tag{14}$$

$$p_y = c_y\left(\frac{n}{c}\right) = L_x\sqrt{\frac{1}{s d_x d_y}}\sqrt{n} \tag{15}$$

where the ratio n/c is the average number of particles per cell.

As noted earlier, domain decomposition is accomplished by a recursive bisection technique in which the aspect ratio of the domain determines whether the cut is made vertically or horizontally. For instance, if the aspect ratio exceeds one, the cut minimising the interface length, and hence communication costs, is a vertical one. Figure 9 shows decomposition strategies for 2, 4 and 8 servers that minimise the interface length for aspect ratios greater than one. For the two server case, the number of particles to be transmitted for one of the subdomains is simply p_y. For the four server case, that number grows on average to $3p_y/2$ when the aspect ratio exceeds 2, and $(p_x + p_y)/2$ otherwise. For the eight server case, the number is $7p_y/4$ when the aspect ratio exceeds 4, and $(3p_y + p_x)/4$ otherwise. Again, each of these expressions is the average

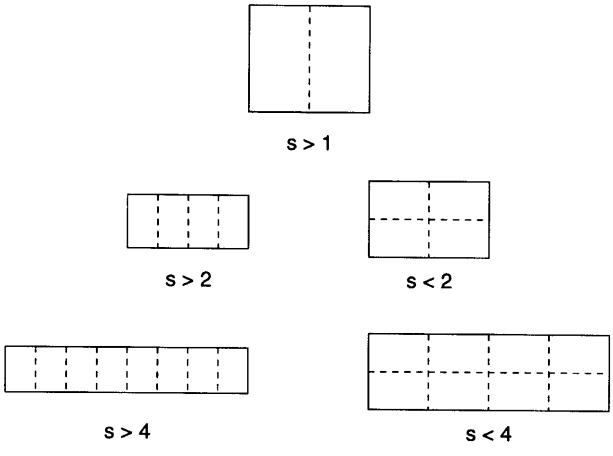

Fig. 9. Domain decomposition for various aspect ratios.

number of particles transmitted for a single subdomain. Using the client-server model, as we have done, each server both sends (inner) and receives (outer) boundary particles. Therefore, combined with the memory required to represent and transmit a single particle, the round-trip communication model shown earlier in Fig. 3 can be used to approximate the total cost of communication.

6.3. Distributed Computation and Communication Costs

Using the sequential computation and communication costs, we derive expressions for the distributed costs for 2, 4 and 8 servers when the aspect ratio of the domain is unity or greater.

Beginning with communication costs, we make use of the empirical results presented in Section 3.3 on Ethernet performance characteristics, where round-trip cost is expressed as a linear function of message size in bytes for N servers

$$T^N_{comm}(x) = m_N x + b_N \tag{16}$$

Those same costs can also be expressed in terms of the number of particles communicated, since each requires the same amount of memory, as

$$T^N_{comm}(p) = m_N p g + b_N \tag{17}$$

where the state of each particle is represented by g bytes of information (88 bytes in ours).

At each time step, particles are transmitted from each server to the client, and from the client back to the servers. The server-to-client communication includes particles that comprise the inner boundary and those that have exited the spatial region assigned to the server. The client-to-server communication includes particles that comprise the outer boundary and those that have entered the spatial region assigned to the server. While we have quantified the number of boundary particles above, the number of migrating particles cannot be accurately estimated. Hence, for the latter we include only the constant term of the communication cost equation.

With respect to computational costs, if we assume perfect load balancing, the total cost is simply T_{seq}/N, where N is the number of servers. Then, for an aspect ratio of one or greater, the overall costs for performing a distributed simulation can be approximated as

$$T^2_{dist} = \frac{T_{seq}}{2} + m_2 p_y g + 2b_2 \tag{18}$$

$$T^4_{dist} = \frac{T_{seq}}{4} + m_4 \min\left(\frac{3p_y}{2}, \frac{p_x + p_y}{2}\right) g + 2b_4 \tag{19}$$

$$T^8_{dist} = \frac{T_{seq}}{8} + m_8 \min\left(\frac{7p_y}{4}, \frac{3p_y + p_x}{4}\right) g + 2b_8 \tag{20}$$

As noted earlier, the T_{seq} terms (computation) grow as $O(n)$ while the p_x and p_y terms (communication) grow only as $O(\sqrt{n})$. These expressions include the following approximations:

- The initial, structured arrangement of particles is assumed constant, even though particles move throughout the simulation, causing variations in the amount of computational work within the cells and hence load imbalances.
- The cost of communicating interface boundaries is included, but not the time required to marshall and unmarshall particle collections into buffers before and after transmission.
- Only the constant 'start-up' term for communicating migrating particles is included, since it is difficult to predict the linear term.

7. Timing Study

The overall performance of a computer code is best gauged by actual timing measurements. These must be made and interpreted with care, however, since they not only depend upon the algorithms used, but also on other factors, such as the underlying hardware, effectiveness of the compiler, the efficiency of library routines, and the manner in which resources are allocated by the operating system, e.g. paging in a virtual memory system. Also of concern in our implementation, as we have seen, are communication overheads in a distributed processing environment.

7.1. Test Problem

Figure 10 depicts a plate impact problem in which a solid nickel (*Ni*) flyer strikes a target plate of

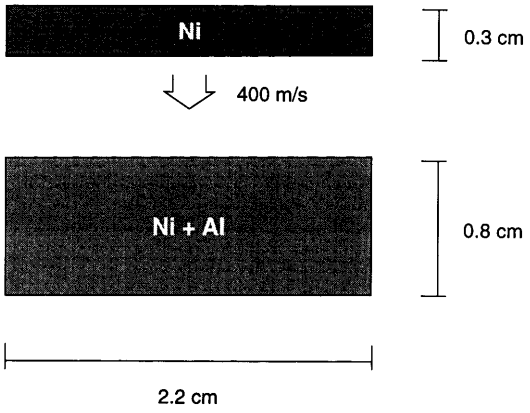

Fig. 10. Plate impact problem.

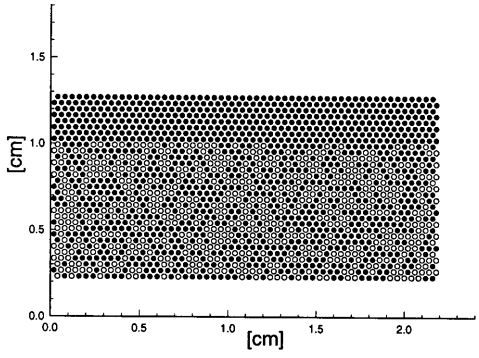

Fig. 11. Initial geometry.

nickel and aluminum ($Ni + Al$) in equal volumetric fractions [25]. The flyer has an initial velocity of 400 m/s and the target is at rest. The thickness of flyer and target are, respectively, 0.3 and 0.8 cm, and the plates have a common width of 2.2 cm.

A discrete element representation of the problem appears in Fig. 11, in which 1689 particles, aligned on a diagonal, are used to represent the combined plates. Particles have a radius of 0.02 cm with a spacing d_x of 0.04 cm and d_y of 0.03464 cm. The maximum interaction radius R_c is 0.1 cm, and the neighbour interaction radius $R_c + \Delta R$ is chosen to be 0.11 cm. Using a minimal cell size the total number of cells is 171, with $c_x = 19$ and $c_y = 9$.

After 200 timesteps, the particle positions 5 μs after impact are shown in Fig. 12. Qualitatively, the flyer and target have separated, and spallation has occurred in the target plate. Additional details of the simulation conducted with the original DM^2 code, including velocity profiles and verification with experimental data, can be found elsewhere [25].

7.2. Timing Results

Executing the small test problem above takes only a minute or two on a single machine. However, our

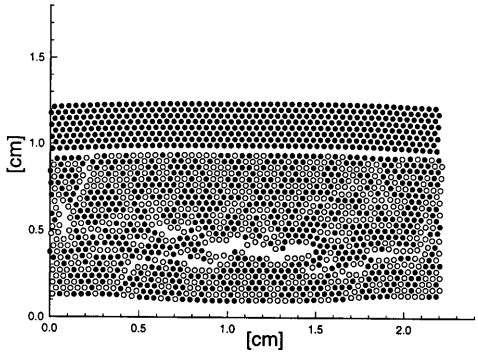

Fig. 12. Geometry at 5 μs.

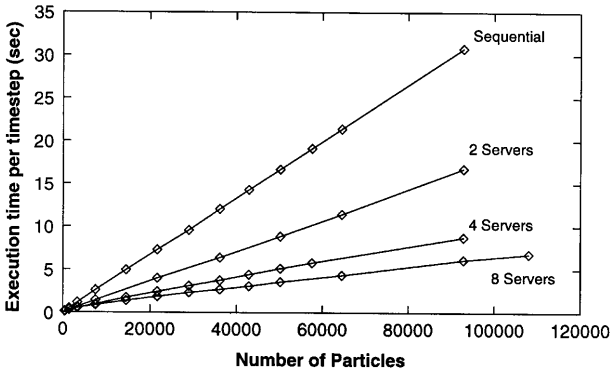

Fig. 13. Execution time of sequential and distributed implementations.

interest in improving the performance of DM^2 is so we can simulate models with hundreds of thousands of particles or more. To assess the performance of the implementation, and to validate our performance model, we have run numerous simulations on various problems, and here present results that were obtained on the test problem above.

Results were obtained by varying the number servers and the number of particles in the simulation while keeping constant the aspect ratio (which we set to 1.3571 for the following). To obtain representative timings, in all cases we report an average time per timestep, with the total time itself being an average of at least three runs, which were typically very similar. Figure 13 shows the execution time of several runs with different numbers of particles. Both the sequential and distributed implementations have linear performance, and as expected, we see improvement by increasing the number of servers that are used.

The speedup, or ratio of sequential to distributed execution times, is shown in Fig. 14 for the same runs, where the unadorned curves are from the performance model previously derived. Looking first

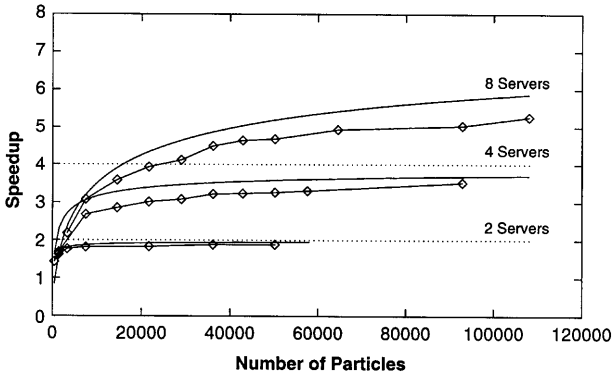

Fig. 14. Speedup versus number of particles.

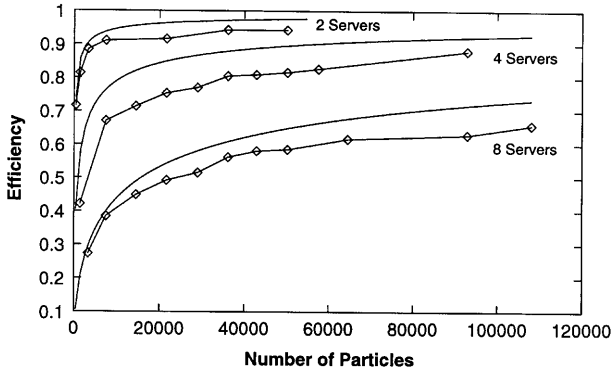

Fig. 15. Efficiency versus number of particles.

at actual performance, we see that, as expected, the speedup obtained on small problems is modest, but on larger ones is quite impressive given the challenges of both problem and computing environment. With respect to the problem, short-range interaction models, as we have noted, are of finer granularity than long-range ones. With respect to the environment, fine-to-medium grained parallelism can present a challenge for high-latency networks, in this case a 10 Mbps Ethernet. Looking at theoretical performance, we see, as expected, an over-estimate due to several factors mentioned earlier, including load imbalances and terms dropped from the communication cost model. Nevertheless, there is very good qualitative agreement, and the quantitative agreement is reasonable enough for making predictions about performance.

The corresponding efficiency curve, the ratio of speedup to the number of servers, is shown in Fig. 15. As predicted by the performance model, for a fixed number of servers, the efficiency improves with problem size since the computation cost grows as $O(n)$ and the communication cost grows only as $O(\sqrt{n})$. However, for a fixed problem size, the use of additional servers reduces overall efficiency, as expected, and this is observed in Fig. 16.

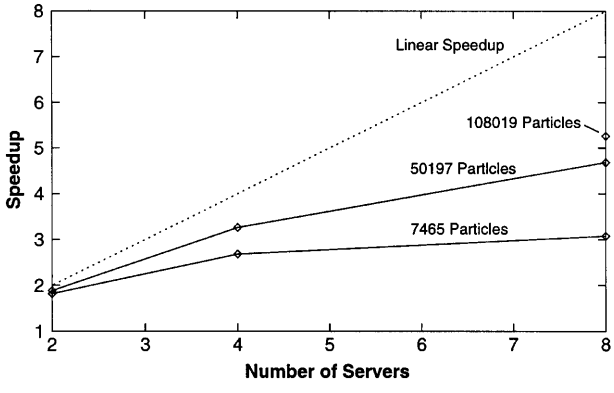

Fig. 16. Ideal and observed speedups.

7.3. Effects of Changes to the Environment

The results presented above were obtained with a client-server implementation on a cluster of Sun Ultra-10 Workstations on a 10BaseT Ethernet. How would the implementation fare in another setting? Using our performance model, we show the effects of several environment variations in Fig. 17, plotting speedups for the eight-server case. First, note the speedup predicted in the current environment. What happens when, all other things being equal, computers are upgraded and a factor of ten improvement in *processing* speed is obtained? The bottom curve of the figure shows that the distributed implementation is of little if any use, and may actually be worse than a sequential implementation. While results will be obtained more quickly by upgrading computers, the inefficiency of the distributed implementation makes it completely impractical.

Going in the other direction, we consider the effect of reducing communication costs by switching from a client-server model to a *peer-to-peer* model. By 'peer-to-peer' we mean a strategy in which server processes communicate directly with each other, eliminating the client as the middle-man. Although somewhat more difficult to develop, a good peer-to-peer implementation could reduce communication costs by a half, thereby improving the overall efficiency of the code.

The effect of a different improvement, a change in the communication environment, can also estimated. The 100BaseT standard (IEEE 802.3u), also known as Fast Ethernet, might be used to improve communication by approximately a factor of ten, yielding very impressive speedups for even small problems. Going one step further to Gigabit Ethernet, the recently adopted 1000BaseT standard (IEEE 802.3ab), would improve communication by

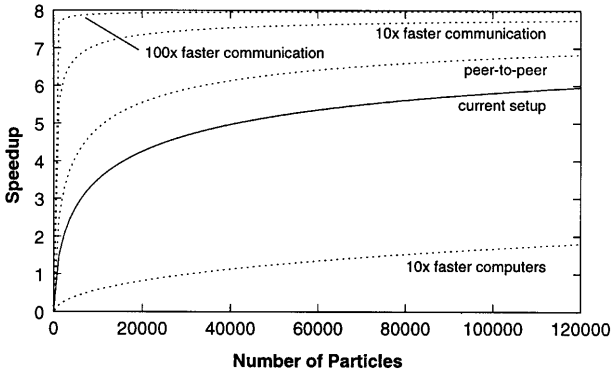

Fig. 17. Variations in architecture and hardware performance for eight servers.

approximately another factor of ten, resulting in a near-perfect linear speedup for problems of any size.

8. Closure

This study shows that particle methods with short-range interaction forces, some of the most difficult to parallelise, can be implemented to execute efficiently on stock network hardware. Our distributed implementation based on DM^2, a discrete element code originally developed at NCSU and Los Alamos National Laboratory, obtains good speedups with eight processors even on a conventional, non-dedicated network of workstations. As problem sizes get larger its performance improves since the computation-to-communication ratio grows with n, the number of particles, as $O(\sqrt{n})$. We have used this implementation to simulate systems with as many as 200,000 particles using eight processors. Because the computation and storage requirements increase only linearly with the number of particles, it is apparent that improvements in computing power will lead to substantially larger simulations based on this technique in the future.

By developing a simple performance model, we also show the effects of several environment changes on the efficiency of our implementation. Although our studies were performed on a relatively low-end 10BaseT Ethernet, we show that faster networks, e.g. Fast or Gigabit Ethernet, or other technologies such as ATM switches, are even more efficient, and would enable a substantially larger number of servers to contribute effectively to solve large-scale discrete element problems.

There are certainly opportunities for improving our implementation. One of the most obvious is in replacing the client-server model with peer-to-peer communication, which would reduce communication costs by as much as a factor of one half. Other efficiency improvements, e.g. optimising particle memory requirements and tuning communication protocols, would also be of benefit. Finally, to enhance usability, the implementation should provide a simple recovery mechanism for better tolerating hardware faults, and a dynamic load balancing scheme for problems that experience large particle movements.

Acknowledgements

The authors would like to thank Yasuyuki Horie and Kazushige Yano of Los Alamos National Laboratory for their help with discrete element modeling and the DM^2 code.

References

1. Plimpton, S. (1995) Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics, 117, 1–19
2. Tang, Z. P., Horie, Y., Psakhie, S. G. (1995) Discrete meso-element modeling of shock processes in porous materials. Vol. 1: Theory and model calculations. Technical Report, North Carolina State University, Raleigh, NC
3. Carrillo, A. R., Horner, D. A., Peters, J. F., West, J. E. (1996) Design of a large scale discrete element soil model for high performance computing systems. Proceedings, Supercomputing '96, IEEE Computer Society Press, Washington, DC (http://www.supercomp.org/sc96)
4. Hockney, R. W., Eastwood, J. W. (1988) Computer Simulation Using Particles. Adam Hilger Publishing, Philadelphia
5. Alder, B. J., Wainwright, T. E. (1959) Studies in molecular dynamics I. General Method. Journal of Chemical Physics, 31, 459–466
6. Cundall, P. A. (1971) A computer model for simulating progressive, large-scale movements in block rock systems. Proceedings Symp. Int. Soc. Rock Mech., Nancy, 11, Art. 8
7. Christoffersen, J., Mehrabadl, M. M., Nemat-Nasser, S. (1981). A micromechanical description of granular material behavior. Journal of Applied Mechanics, 48, 339–344
8. Bathurst, R. J., Rothenburg, L. (1988) Micromechanical aspects of isotropic granular assemblies with linear contact interactions. Journal of Applied Mechanics, 15, 17–23
9. Zhang, R. (1993) Analysis of hydraulic problems using the discrete element method. Master's thesis, Colorado School of Mines, Golden, CO
10. Meegoda, N. J., Chang, K. G. (1994) Modeling of viscoelastic behavior of hot mix asphalt (HMA) using discrete element method. In: Basham, K. D. (Editor), Proceedings of the 3rd ASCE Materials Engineering Conference, Infrastructure: New Materials and Methods of Repair, San Diego, CA, 804–811
11. Meegoda, N. J., Washington, D. (1994) Massively parallel computers for microscopic modeling of soils. In: Siriwardane, H. J., Zaman, M. M. (Editors), Computer Methods and Advances in Geomechanics, A. A. Balkema Publishers, 617–622
12. Barnes, J. E., Hut, P. (1986). A hierarchical $O(n \log n)$ force-calculation algorithm. Nature, 324, 446–449
13. Greengard, L., Rokhlin, V. (1987). A fast algorithm for particle simulations. Journal of Computational Physics, 73, 325–348
14. Verlet, L. (1967) Computer experiments on classical fluids: I. Thermodynamical properties of Lennard-Jones molecules. Phys Rev, 159, 98–103
15. Hockney, R. W., Goel, S. P., Eastwood, J. W. (1974) Quiet high-resolution computer models of a plasma. Journal of Computational Physics, 14, 148–158

16. Fox, G. et al. (1988) Solving Problems on Concurrent Processors. Vol. 1, Prentice-Hall
17. Tamayo, P., Giles, R. (1992) A parallel scalable approach to short-range molecular dynamics on the CM-5. Proceedings, Supercomputing '92, IEEE Press, Washington, DC, 240
18. Pinches, M. R. S., Tildesley, D. J., Smith, W. (1991) Large-scale molecular dynamics on parallel computers using the link-cell algorithm. Molecular Simulation, 6, 51
19. Tang, Z. P., Horie, Y., Psakhie, S. G. (1997) Discrete meso-element modeling of shock processes in powders. In: Davison, L., Horie, Y., and Shahinpor, M. (Editors), High-Pressure Shock Compression of Solids IV. Springer, New York, 143–176
20. Anderson, T. E., Culler, D. E., Patterson, D. A. (1995) A Case for NOW (Networks of Workstations). IEEE Micro, 15(1), 54–64
21. Baugh, J. W., Jr., Sharma, S. K. (1994) Evaluation of distributed finite element algorithms on a workstation network. Engineering with Computers, 10(1), 45–62
22. Chadha, H. S., Baugh, J. W., Jr. (1996) Network-distributed finite element analysis. Advances in Engineering Software, 25, 267–280
23. Baugh, J. W., Jr., Kakivaya, G. R., Stone, J. R. (1998) Intractability of the dial-a-ride problem and a multiobjective solution using simulated annealing. Engineering Optimization, 30(2), 91–123
24. Loughlin, D. H., Ranjithan, S., Baugh, J. W., Jr., Brill, E. D., Jr. (2000) Application of genetic algorithms for the design of ozone control strategies. Journal of the Air & Waste Management Association, in press
25. Yano, K., Schwarz, O. J., Tang, Z. P., Horie, Y. (1996) Discrete meso-element modeling of shock processes in porous materials. Vol. 2: User's Manual for the DM^2 Code. Technical Report, North Carolina State University, Raleigh, NC