M.I.T. Media Lab Vision and Modeling Group Technical Report No. 189, July 1992
To appear: 1992 Eurographics Workshop on Animation and Simulation,Cambridge, UK

DISTRIBUTED PHYSICAL SIMULATION

Martin Friedmann and Alex Pentland
Vision and Modeling Group, The Media Laboratory
Massachusetts Institute of Technology
20 Ames Street, Cambridge MA 02139, U.S.A.

ABSTRACT
We present a system for distributed real-time execution of non-rigid physical simulations whose demonstrated efficiency increases nearly linearly as a function of the number of processors. To achieve this scaling behavior one must minimize network and processor contention; in our system this is made possible by replacing synchronous operation by the weaker condition of of *bounded asynchrony*, and by use of compact modal representations of shape and deformation. The system allocates computational resources among networked workstations using a simple, efficient "market-based" strategy, thus avoiding the problems of central control.

1 Introduction

The prospect of having simulated environments on computers, in which multiple users can move and interact comfortably, has excited researchers for years. The possible applications for real-time physical simulation include visual information environments, CAD/CAM with interactive stress testing, distributed simulations, and physically-based animation. Unfortunately, tremendous computational resources are required to compute object dynamics, detect collisions, calculate friction, etc., in order to simulate complex multibody environments in interactive time. Consequently, most physically-based modeling and simulation has been confined to the domain of batch processing.

For applications not requiring physical simulation, large scale computer-simulated environments are already in operation. Simulation Network (SIMNET) [14], a nation-wide effort in the U.S.A., is a large scale "virtual reality" carried out over a standard computer network involving up to several hundred computer workstations. It is a distributed system that can be scaled up to handle very large, complex environments and hundreds of users in interactive time.

SIMNET, however, was not designed to be capable of providing physically-based simulation on a large scale (or even to be capable of running on vendor supplied hardware). Users only rove around and view each other from their own positions, and do not interact physically with each other or with their environment. SIMNET has, however, demonstrated that current-day network communications, computer workstations, and graphics hardware are capable of supporting large simulated environments. Also, there are software efforts presently underway to port SIMNET functionality to standard graphics hardware, and to include physical simulation capabilities [5, 18]. The goal of this work is to effectively address fundamental problems facing these efforts.

The concept of extending algorithms into the domain of distributed processing is not, of course, new to the graphics community. A wide variety of distributed algorithms exist for high quality and real-time image rendering. The goal of this work is similar; to obtain large, physically-based simulated environments which run in near real-time. Instead of having one central workstation simulate all moving objects and broadcast updates to other workstations for display, each processor becomes responsible for simulating a small subset of moving objects within a larger set of apparently static objects. These surrounding "static" objects are also moving and deforming, of course, but their behavior is being simulated in parallel by the other processors.

The result of distributing the computational load can be a large performance improvement. Unfortunately, a straightforward implementation of distributed physical simulation produces little speedup, because of network overhead and network/processor contention. Consequently, to achieve substantial gains from a parallel implementation we must first solve several problems:

1.1 Too Much Data

Non-rigid simulations typically require changing every polygon vertex, however such large amounts of data cannot be broadcast sufficiently quickly over current networks. Consequently, a concise description of non-rigid behavior and geometry is required.

Our solution is to adapt the technique of *modal dynamics* [11, 15, 6] to the realm of parallel and distributed processing [4]. All rigid and non-rigid motion can be described by the linear superposition of the object's *free vibration modes*. Normally only a few such modes are required to obtain very accurate shape descriptions; in fact, vibration modes are the optimally compact description assuming that external forces are uniformily distributed.

By using a modal description framework, we can avoid broadcasting polygons and instead broadcast only a few modal coefficients. In our system this typically reduces network traffic by more than an order of magnitude.

1.2 Synchronous Operation

Physical simulation requires that all of the processors must remain at least approximately in lockstep, so that no part of the physical simulation lags behind some other part. Unfortunately, this means that all of the processors are trying to communicate with each other at the same time, leading to significant network contention. As a consequence, a large fraction of each processor's time is wasted while waiting for updates to arrive.

To solve this problem we introduce the concept of *bounded asynchrony*, and prove that a limited amount of asynchrony can be allowed without degrading overall simulation accuracy. This small amount of asynchrony allows interprocessor communication to proceed more or less continuously, significantly reducing the problems of network and processor contention.

1.3 Variable Loading

The computational load imposed by physical simulation varies over time, due to collisions, changing geometry, and external factors. This causes difficult problems concerning resource allocation and task migration.

To address this problem we have adopted an approach based on use of simple localized mechanisms such as price, trade, and competition. Such an approach is known as an *agoric system* [10], and has been shown capable of combining local decisions made by separate agents into globally effective behavior. We have found that in our application such simple mechanisms perform quite satisfactorily.

The following sections of this paper will address each of these problems in turn, presenting our solutions in more detail. The final section presents experimental results.

2 Communications

Standard physical simulation requires that the state of all objects be updated in lockstep. In a distributed system this requires all processors to wait while they trade state changes between steps of simulation — a serious loss of efficiency. Similarly, standard physical simulation of nonrigid objects requires modification of all polygon vertices. In a straightforward distributed implementation, this requires large amounts of data to be traded among all processors, quickly saturating the network.

Two conditions required for this communications scheme to be successful are that the packet size be small, so that network bandwidth is not exceeded, and that state changes be asynchronous, so that network contention is minimized. Unfortunately, both of these conditions are at odds with the standard methods for physical simulation.

Our solution to the network saturation problem is to employ a concise parametric representation to represent nonrigid object deformations. In most situations this allows us to avoid sending polygon vertices between machines, and instead send only a short parametric description that converts the old shape into the new deformed shape. The use of a parametric description of shape also has the advantage that it can be used to obtain a generalized implicit function representation [15], thus allowing fast collision detection and characterization [17]. This approach is described further in the next section.

The second part of our solution to the problem of network saturation is to allow *bounded asynchrony* in the simulation. By allowing a small amount of asynchrony in our physical simulations, we have been able to simultaneously improve both the accuracy and the efficiency of the system. This is described in section 4.

3 Parametric Representation

The physical simulation algorithm used in our implementation is a modal representation of nonrigid deformation (see references [11, 15, 6], the most up to date work being presented by Irfan Essa in the main session of Eurographics '92). The modal representation is the optimally efficient parameterization over the space of all physical deformations, as it is the Karhunen-Loeve expansion of the object's stiffness matrix. However any set of standard parametric deformations [1] can be used as long as they span the space of nonrigid deformations expected, and can be calculated from the results of the physical simulation. An example of describing nonrigid physical behavior using standard deformations can be found in reference [16].

3.1 Modes as a Parametric Representation

In the finite element method (FEM), energy functionals are formulated in terms of nodal displacements $\mathbf{U}$, and iterated to solve for the nodal displacements as a function of impinging loads $\mathbf{R}$:

$$\mathbf{M}\ddot{\mathbf{U}} + \mathbf{C}\dot{\mathbf{U}} + \mathbf{K}\mathbf{U} = \mathbf{R} \tag{1}$$

This equation is known as the FEM *governing equation*, where $\mathbf{U}$ is a $3n$ x 1 vector of the $(\Delta x, \Delta y, \Delta z)$ displacements of the n nodal points relative to the object's center of mass, $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{K}$ are $3n$ by $3n$ matrices describing the mass, damping, and material stiffness between each point within the body, and $\mathbf{R}$ is a $3n$ x 1 vector describing the x, y, and z components of the forces acting on the nodes.

To obtain a physical simulation, one integrates Equation 1 using an iterative numerical procedure at a cost proportional to the stiffness matrices' bandwidth. To reduce this cost we can transform the problem from the original nodal coordinate system to a new coordinate system whose basis vectors are the columns of an n x n matrix $\mathbf{P}$. In this new coordinate system the nodal displacements $\mathbf{U}$ become *generalized displacements* $\tilde{\mathbf{U}}$:

$$\mathbf{U} = \mathbf{P}\tilde{\mathbf{U}} \tag{2}$$

Substituting Equation 2 into Equation 1 and premultiplying by $\mathbf{P}^T$ transforms the governing equation into the coordinate system defined by the basis $\mathbf{P}$:

$$\tilde{\mathbf{M}}\ddot{\tilde{\mathbf{U}}} + \tilde{\mathbf{C}}\dot{\tilde{\mathbf{U}}} + \tilde{\mathbf{K}}\tilde{\mathbf{U}} = \tilde{\mathbf{R}} \tag{3}$$

where

$$\tilde{\mathbf{M}} = \mathbf{P}^T\mathbf{M}\mathbf{P}; \quad \tilde{\mathbf{C}} = \mathbf{P}^T\mathbf{C}\mathbf{P}; \quad \tilde{\mathbf{K}} = \mathbf{P}^T\mathbf{K}\mathbf{P}; \quad \tilde{\mathbf{R}} = \mathbf{P}^T\mathbf{R} \tag{4}$$

With this transformation of basis, a new system of stiffness, mass, and damping matrices can be obtained which has a smaller bandwidth then the original system.

The optimal basis $\mathbf{\Phi}$ has columns that are the eigenvectors of both $\mathbf{M}$ and $\mathbf{K}$ [2]. These eigenvectors are also known as the system's *free vibration modes*. Using this transformation matrix we have

$$\mathbf{\Phi}^T\mathbf{K}\mathbf{\Phi} = \mathbf{\Omega}^2, \qquad \mathbf{\Phi}^T\mathbf{M}\mathbf{\Phi} = \mathbf{I} \tag{5}$$

where the diagonal elements of $\mathbf{\Omega}^2$ are the eigenvalues of $\mathbf{M}^{-1}\mathbf{K}$ and remaining elements are zero. When the damping matrix $\mathbf{C}$ is restricted to be *Rayleigh damping*, then it is also diagonalized by this transformation.

The lowest frequency vibration modes of an object are always the rigid-body modes of translation and rotation. The next-lowest frequency modes are smooth, whole-body deformations that leave the center of mass and rotation fixed. Compact bodies — solid objects like cylinders, boxes, or heads, whose dimensions are within the same order of magnitude — normally have low-order modes which are intuitive to humans: bending, pinching, tapering, scaling, twisting, and shearing. Some of the low-order mode shapes for a cube are shown in Figure 1. Bodies with very dissimilar dimensions, or which have holes, etc., can have very complex low-frequency modes.

The bandwidth advantages of using such a parametric representation can be enormous. In a typical case in our system, the complete parametric representation of shape and nonrigid

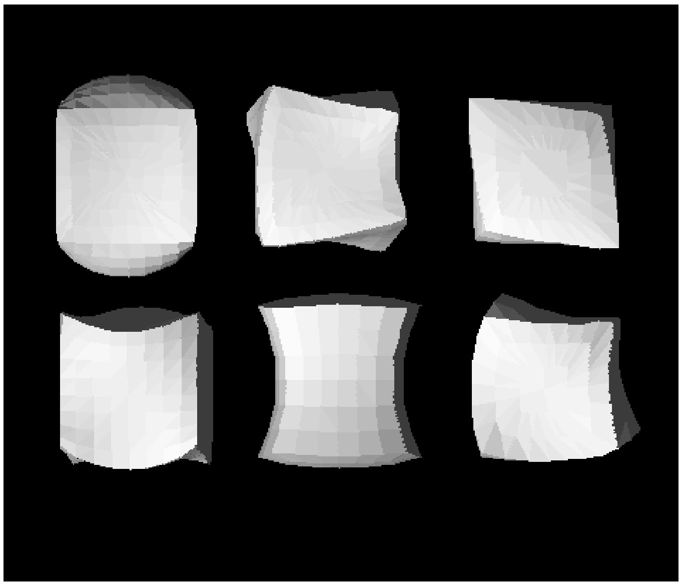


FIGURE 1. A few of the vibration mode shapes of a 27 node isoparametric element.

behavior requires only 120 bytes, whereas the vertex, polygon and normal representation with no non-rigid behavior information requires at least 4704 bytes.

3.2 Combination with Implicit Function Geometry

A further advantage is that such parametric representations may be combined with an implicit function surface to obtain extremely efficient collision detection [15, 11, 6]. In object-centered coordinates $\mathbf{r} = [r, s, t]^T$, the implicit equation of a spherical surface is

$$f(\mathbf{r}) = f(r, s, t) = r^2 + s^2 + t^2 - 1.0 = 0.0 \tag{6}$$

This equation is also referred to as the surface's *inside-outside function*, because to detect contact between a point $\mathbf{X}_p = [X_p, Y_p, Z_p]^T$ and the volume bounded by this surface, one simply substitutes the coordinates of $\mathbf{X}$ into the function f. If the result is negative, then the point is inside the surface. Generalizations of this basic operation may be used to find line-surface intersections or surface-surface intersections.

A solid defined in this way can be easily positioned and oriented in global space, by transforming the implicit function to global coordinates, $\mathbf{X} = [X, Y, Z]^T$ we get [15]:

$$\mathbf{X} = \mathcal{R}\mathbf{r} + \mathbf{b} \tag{7}$$

where $\mathcal{R}$ is a rotation matrix, and $\mathbf{b}$ is a translation vector. The implicit function's positioned and oriented (rigid) inside-outside function becomes (using Equation 7):

$$f(\mathbf{r}) = f(\mathcal{R}^{-1}(\mathbf{X} - \mathbf{b})). \tag{8}$$

Any set of implicit shape functions can be generalized by combining them with a set of global deformations $\mathcal{D}$ with parameters $\mathbf{m}$. For particular values of $\mathbf{m}$ the new deformed surface is defined using a deformation matrix $\mathcal{D}_\mathbf{m}$:

$$\mathbf{X} = \mathcal{R}\mathcal{D}_\mathbf{m}\mathbf{r} + \mathbf{b} \tag{9}$$

In our system the deformations used are the *modal shape polynomial functions*, defined by transforming the original finite element shape functions to the modal coordinate system (see [13]). These polynomials are a function of $\mathbf{r}$, so that Equation 9 becomes:

$$\mathbf{X} = \mathcal{R}\mathcal{D}_{\mathbf{m}}(\mathbf{r})\mathbf{r} + \mathbf{b} \tag{10}$$

Combining the non-rigid deformation of Equation 10 with this inside-outside function we obtain,

$$f(\mathbf{r}) = f(\mathcal{D}_{\mathbf{m}}^{-1}(\mathbf{r})\mathcal{R}^{-1}(\mathbf{X} - \mathbf{b})) \tag{11}$$

This inside-outside function is valid as long as the inverse polynomial mapping $\mathcal{D}_{\mathbf{m}}^{-1}(\mathbf{r})$ exists. In cases where a set of deformations has no closed-form inverse mapping, Newton-Raphson and other numerical iterative techniques have to be used.

This method of defining geometry, therefore, provides an inherently more efficient mathematical formulation for contact detection than geometric representations such as polygons or splines.

3.3 Additional Considerations for Displaying Graphics

Virtually all modern 3-D graphics libraries, such as the Programmer's Hierarchical Interactive Graphics System (PHIGS/PHIGS+), have only simple linear transformation capabilities, in order to allow for translation, scaling and perspective viewing of scenes composed of rigid objects. Because these library standards are optimized for storing static objects in a fixed display list they are poorly suited to the task of interactively displaying non-rigid simulations. Consequently, when using Equation 10, the entire vertex/polygon representation of all moving objects must be uploaded from the simulation engine into the graphics display list at each frame update.

But by considering only the whole-body deformations which are not functions of $\mathbf{r}$, namely those of linear scale and shear, we are able to, as in Equation 9, compose a single $4x4$ transformation matrix describing both the rigid and non-rigid motion of the simulated object. This allows us to preload the object's undeformed shape $\mathbf{r}$ into the display list, and obtain impressively fast displays of non-rigid dynamic simulation using standard graphics software and hardware.

To accomplish this we restrict the object's parameters of deformation to be
$\mathbf{m} = [Scale_x, Scale_y, Scale_z, Shear_x, Shear_y, Shear_z]$ giving the deformation matrix,

$$\mathcal{D}_{\mathbf{m}} = \begin{bmatrix} Scale_x & Shear_z & Shear_y \\ Shear_z & Scale_y & Shear_x \\ Shear_y & Shear_x & Scale_z \end{bmatrix}. \tag{12}$$

This restricted deformation matrix also simplifies the computation necessary for collision detection in Equation 11, because the object's global inverse deformation matrix $\mathcal{D}_{\mathbf{m}}^{-1}(\mathbf{r})$ is a single matrix that is not dependent on $\mathbf{r}$.

4 Bounded Asynchronous Operation

Synchronizing physical simulations of heterogeneous objects and situations intrinsically requires wasting a substantial amount of network bandwidth and CPU processing time. This is because each processor must wait until every other processor has broadcast its results the previous time step before it can begin processing the next. Consequently, it is extremely desirable to allow some sort of asynchronous operation.

In the following, we mathematically analyze synchronous and asynchronous operation modes by proving that the operation of an asynchronous network is equivalent to that of a particular type of synchronous network whose update rule considers the Hessian of the energy function. This result will then allow us to formulate a type of bounded asynchronous operation that results in faster and more stable network performance.

4.1 Energy Minimization

Physical simulation can be stated as a energy minimization problem. A time-varying three-dimensional potential field is defined by the sum of gravity, collisions, internal elasticity and damping, as well as the artificial potentials contributed by user-defined constraints. Objects act to minimize their potential energy by descending along the gradient of this potential energy surface.

Designating the system state by the vector $\mathbf{U}(T)$ and the potential field by the scalar-valued function $E(\mathbf{U}(T))$, then the system evolution obeys:

$$\dot{\mathbf{U}}(T) = -\nabla_{\mathbf{U}} E(\mathbf{U}(T)) \tag{13}$$

The system's state parameters "roll" down this energy surface until a collision occurs or a local minimum or "rest state" is reached. When a collision occurs, the potential energy surface warps in response to the new forces acting upon the colliding objects. The state parameters of these objects then begin to evolve along the new gradient direction.

To simulate this physical behavior on a computer, we discretize time into units ΔT, calculate the energy gradient of the entire physical system as a function of its parameters $\mathbf{U}$ at each time step, and then iteratively update the system parameters along the gradient direction by an amount proportional to the product of the gradient magnitude and the time step. If we keep ΔT sufficiently small, then we will accurately track the true gradient direction and the simulation will accurately mimic the behavior of a real physical system.

4.2 Equivalent Operation Mode

In the preceding we assumed synchronous operation where the system state $\mathbf{U}(T_j)$ is updated at each time $T_{j+1} = T_j + \Delta T$, for example,

$$\mathbf{U}(T_{j+1}) = \mathbf{U}(T_j) + \Delta\mathbf{U}(T_j) \tag{14}$$

where

$$\Delta\mathbf{U}(T_j) = -\nabla_{\mathbf{U}} E(\mathbf{U}(T_j))\Delta T \tag{15}$$

In asynchronous operation, each processor broadcasts the state changes of the objects it is simulating as soon as it has finished a time step. Thus each processor will receive state changes from other processors during each time interval ΔT. To describe asynchronous operation, therefore, we further divide the time interval ΔT into K smaller steps t_l, such that $t_{l+1} = t_l + \Delta t$ where $\Delta t = \Delta T/K$, $t_0 = 0$ and K is large, thus obtaining the following update equations [8]:

$$\mathbf{U}(T_{j+1}) = \mathbf{U}(T_j) + \Delta\mathbf{U}(T_j) \quad , \tag{16}$$

where $\mathbf{U}$ and $\Delta\mathbf{U}$ are now the *time averaged* update equation, which is related to the detailed behavior of the network by the relations

$$\Delta\mathbf{U}(T_j) = \frac{1}{K}\sum_{l=1}^{K} \Delta\mathbf{U}(T_j + t_{l-1}) \quad , \tag{17}$$

$$\Delta\mathbf{U}(T_j + t_{l-1}) = -\nabla_{\mathbf{U}} E(\mathbf{U}(T_j + t_{l-1}))\Delta t \quad , \tag{18}$$

and

$$\mathbf{U}(T_j + t_l) = \mathbf{U}(T_j + t_{l-1}) + \Delta\mathbf{U}(T_j + t_{l-1}) \quad . \tag{19}$$

Equations (16) to (19) describe “micro-state” updates that are conducted throughout each interval T_j whenever the gradient at subinterval t_l is available, that is, whenever one of the network's processor's broadcasts the state changes of the objects that it is simulating.

4.3 Synchronous Equivalent to Asynchronous Operation

We first define the gradient and Laplacian of E at asynchronous times t_l to be

$$\mathbf{A}_l = \nabla_{\mathbf{U}} E\Big|_{\mathbf{U}=\mathbf{U}(t_l)} \qquad\qquad B_l = \nabla^2_{\mathbf{U}} E\Big|_{\mathbf{U}=\mathbf{U}(t_l)} \tag{20}$$

and define that all subscripts of $\mathbf{A}$, B, and t are taken to be modulo K.

We will next note that the gradient at time t_{l+1} can be obtained by using the gradient at time t_1 and Laplacian at times $t_1 \cdots t_l$ as below:

$$\begin{aligned}
\mathbf{A}_{l+1} &= \mathbf{A}_l + B_l \Delta\mathbf{U}_l && (21)\\
&= \Big(I - \eta\Delta t B_l\Big)\mathbf{A}_l && (22)\\
&= \prod_{k=1}^{l}\Big(I - \eta\Delta t B_k\Big)\mathbf{A}_1 \quad . && (23)
\end{aligned}$$

Assuming that ΔT is small, and thus that Δt is also small, then at time T_j the K-time-step time-averaged gradient is

$$\begin{aligned}
\bar{\nabla}_{\mathbf{U}} E\Big|_{\mathbf{U}=\mathbf{U}(T_j)} &= \frac{1}{K}\sum_{l=1}^{K}\mathbf{A}_l && (24)\\
&\approx \frac{1}{K}\sum_{l=1}^{K}\Big(I - \eta\Delta t\sum_{k=1}^{l} B_k\Big)\mathbf{A}_1 && (25)
\end{aligned}$$

$$\approx \left(I - \frac{\eta\Delta T}{2} B_1\right) \mathbf{A}_1 \tag{26}$$

$$= \left(I - \frac{\eta\Delta T}{2} \nabla^2_{\mathbf{U}} E\bigg|_{\mathbf{U}=\mathbf{U}(T_j)}\right) \mathbf{A}_1 \quad . \tag{27}$$

Thus the time-averaged state update equation for an asynchronous network is

$$\frac{d\mathbf{U}}{dt}\bigg|_{\mathbf{U}=\mathbf{U}(T_j)} \approx -\eta \left(I - \frac{\eta\Delta T}{2} \nabla^2_{\mathbf{U}} E\bigg|_{\mathbf{U}=\mathbf{U}(T_j)}\right) \mathbf{A}_1 \quad . \tag{28}$$

This update function is reminiscent of second-order update functions which take into account the curvature of the potential energy surface by employing the Hessian of the potential energy function [3]:

$$\frac{d\mathbf{U}}{dT} = -\eta(I + \rho\nabla^2_{\mathbf{U}} E)^{-1} \nabla_{\mathbf{U}} E \tag{29}$$

where the identity matrix I is a "stabilizer" that improves performance when the Laplacian is small.

Taking the first-order Taylor expansion of equation (29) about $\nabla^2_{\mathbf{U}} E = 0$, we obtain

$$\frac{d\mathbf{U}}{dT} \approx -\eta(I - \rho\nabla^2_{\mathbf{U}} E) \nabla_{\mathbf{U}} E \tag{30}$$

and setting $\rho = \eta\Delta T/2$ we see that equations (28) and (29) are equivalent (given small ΔT so that the Taylor expansion is accurate).

Thus equation (30) is a *synchronous* second-order update rule that is identical to the time-averaged *asynchronous* update rule of equation (28). This proves that the bounded asynchronous operation of such networks using a gradient update rule is equivalent to using a synchronous second-order update rule that considers the Hessian of the potential energy function.

The first condition required to obtain this equivalence is that the time step ΔT is small enough, so that the approximations of equations (26) and (30) are valid. This condition is required of any physical simulation.

The second condition is that we maintain a condition of *bounded asynchrony*: all parameters $\mathbf{U}$ must be updated once (on average) during every interval ΔT. The assumption of bounded asynchrony, however, requires distributing the computational load so that each object's behavior is, on average, being simulated at the same rate. Such resource allocation is a difficult problem in its own right, and is addressed in the following section.

5 Resource Allocation

Most systems which allocate tasks among multiple processors are based on some sort of cost-benefit analysis. Often such analyses require solving difficult nonlinear optimization problems.

In recent years, however, simple distributed systems that use simple, localized mechanisms such as price, trade, and competition have been found to provide surprisingly good performance. Such systems are dubbed *agoric systems* for the greek word *agora* meaning market place [10]. Agoric systems are able to combine local decisions made by separate agents into globally effective behavior. These market-based ideas work because they do not attempt to formulate the "golden rule," or the "perfect loop," instead they set up a *Computational Ecology* [7], in which the simultaneous decisions of multiple and remote rational actors to converge on a global behavior that is for the common good.

A simple yet powerful example of such a system is the Enterprise system [9]. This system treats idle processors as contractors, and those requesting assistance as clients. Clients broadcast requests for "bids" from contractors on executing specific tasks. These requests contain information about the size and nature of the task at hand. A process server evaluates this information given its ability to handle tasks and replies with a bid corresponding to the estimated completion times for the task, and in a sense conveys the localized price that the client would actually pay for choosing that processor for that task.

To visualize our resource allocation problem, imagine each processor as a contractor who wants to work with others in the colony to complete an infinite series of physical simulation iterations. Each iteration is completed with an integral amount of work from each contractor, and must be completed roughly within the global time interval ΔT in order to maintain the condition of *bounded asynchrony.*

If the i^{th} processor is lagging behind others, so that the time ΔT^i for that processor to complete an iteration of the simulation loop is longer than the average, then the condition of bounded asynchrony is jeopardized. Such delinquent processors must give part of their workload to processors that have below-average completion times, so that the simulation will not fall too far out of lock-step.

To determine whether they are delinquent and how much work they need to offload, all processors compare their iteration timing with the average iteration timing of all processors involved. If a processor P^i has an average elapsed timing that is greater than the average, it will try to offload a fraction W^i of its own work to others where,

$$W^i = \frac{\Delta T^i - \Delta T^{avg}}{\Delta T^i}. \tag{31}$$

But to which processor does this extra work go? And how can one processor offloading work be sure that it is not inadvertently combining with another offloading processor to overburden a below average processor? The key idea behind Huberman's [7] work is that there is no easy answer to these questions. They are the wrong questions, as they assume the existence of some perfect answer or "golden rule." A far more effective solution is to do something simple now, and again later if need be.

In this spirit, our system offloads work at any one interval only to the processor P^{min} with the lowest timing (i.e., the one with the most time to spare). We must, however, be conservative about this: suppose that P^{min} was the only processor with a below-average timing and all other processors P^i have above-average timings. In this case every P^i will transfer some work to P^{min}! To prevent this sort of computational catastrophe, we divide W^i by the total number of other processors $(N-2)$ to obtain the the percentage of work

$$W^i = \frac{\Delta T^i - \Delta T^{avg}}{\Delta T^i (N-2)} \tag{32}$$

that each above-average P^i should give to P^{min}.

To execute this rule each timestep the processor needs to know only how many objects it is currently simulating, their relative computational cost, and the timings of the other processors are. The timings can be easily transferred along with the object updates, adding negligible network bandwidth and communications overhead.

6 Experimental Results

This system has been implemented in C, and runs on DECStation 5000/200 and Sun 4 workstations. The user interface processes are separate C processes and are derived from the ThingWorld modeling system [11, 12, 15, 6]. The underlying network code layer handling connections, byte swapping, and client/server requests is similar to that found in the X Window System. This method was chosen because of its familiarity to users, and because of its demonstrated portability and efficiency.

6.1 Performance versus Number of Processors

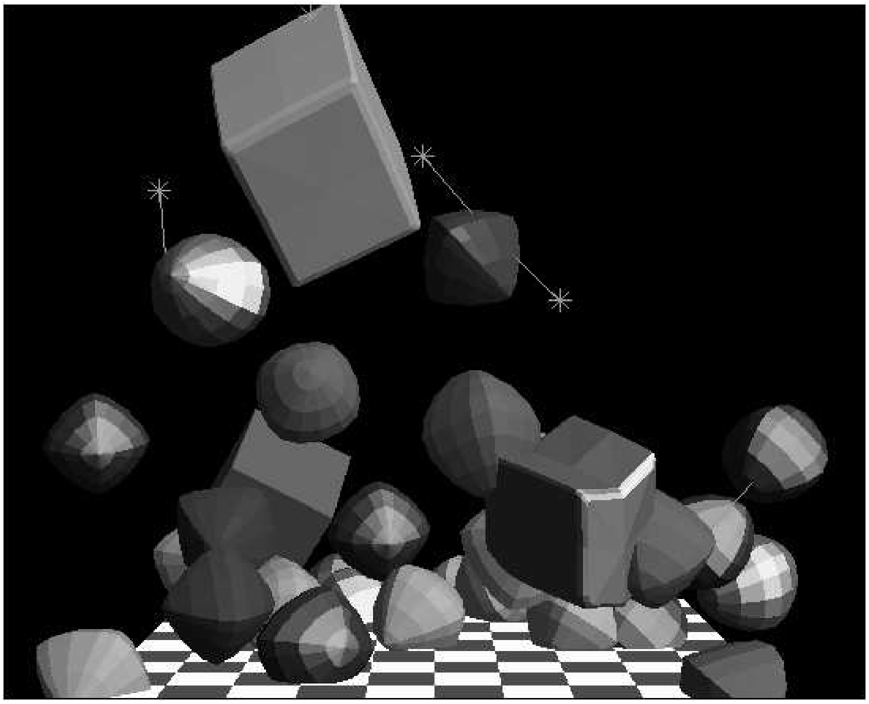
Figure 2(a) shows one frame from a physical simulation of the non-rigid dynamics of 30 objects, including collision detection and response characterization. Thirty modes of deformation were used for each object, and each object's geometry consisted of 144 polygons. Also pictured are some point-position and point-to-point-attachment constraints on the objects being simulated.

Figure 2(b) shows the total elapsed time for 100 time steps of this simulation as a function of the number of processors used and the mode of operation (synchronous versus asynchronous versus ideal scaling). All the processors were of the same type (DECStation 5000/200 workstations), to simplify comparison. Timings do not include graphics processing overhead, including polygon mesh regeneration, z-buffer rendering, and camera transformations, as these costs depend upon camera viewpoint and the specifics of the graphics hardware employed. The average graphics processing overhead in this example was 8.02 CPU seconds over the 100 time steps, so that total execution time for computing and displaying 100 time steps of this physical simulation is the time graphed in Figure 2(b) plus 8.02 seconds.

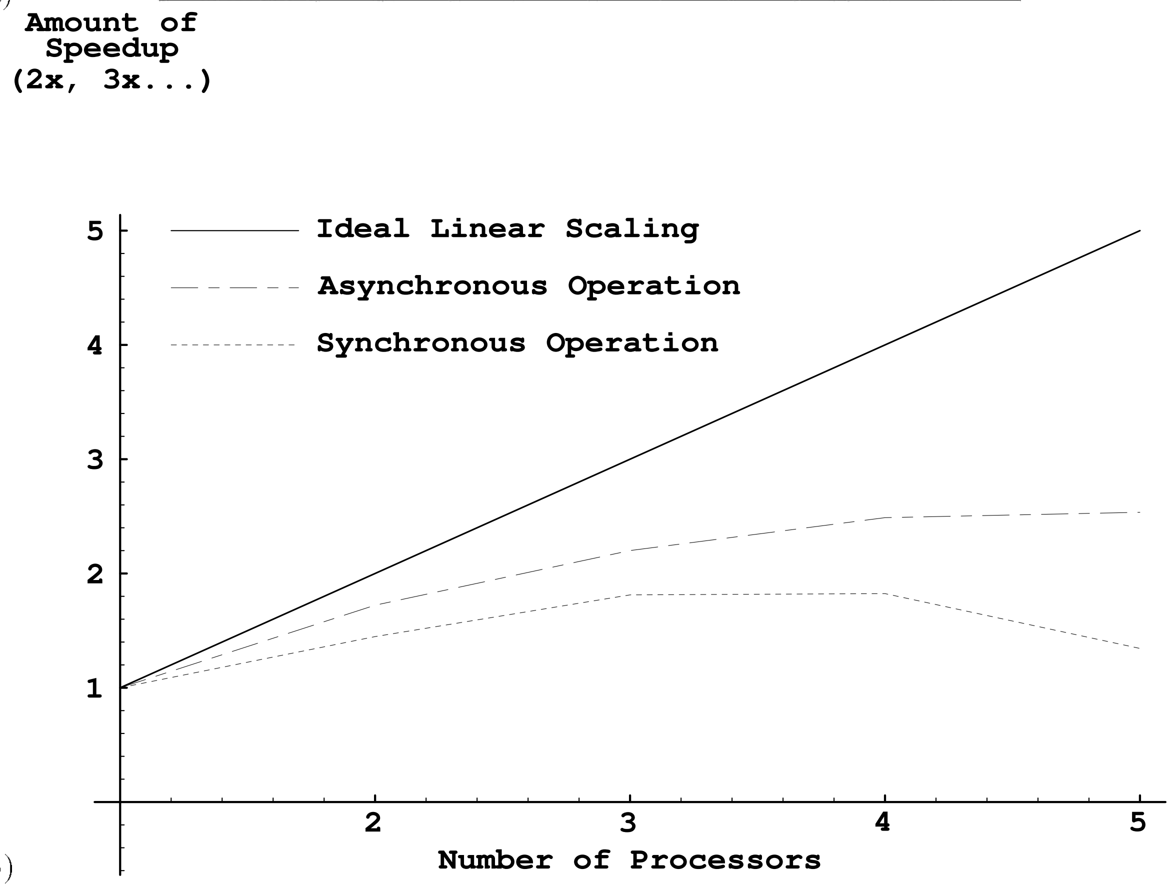
As can be seen, when using an asynchronous update rule the simulation time decreases quite nicely as the number of processors increases. For comparison, the ideal linear scaling function — where total execution time is equal to $1/n$ times the single processor execution time — is plotted as a dashed line. The measured performance of the asynchronous network is qualitatively similar to that of the ideal linear scaling function, with increasing network overhead accounting for most of the differences between ideal and observed execution times.

For comparison, the dotted line in Figure 2(b) shows the performance using a *synchronous* update rule. It can be seen that much worse performance was obtained. In fact, for synchronous operation total execution time actually *increases* with additional processors! This appears to be due to increasing numbers of wait states occurring during the network operation as more and more processors are added.

In our current system performace plateaus between five and ten processors. This is primarily because we do not currently have run with broadcast socket communications. Rather, we still maintain separate socket connections between each pair of machines, a overhead



(a)



(b)

FIGURE 2. Non-rigid physical simulation performance versus number of processors: (a) one frame from the simulation, (b) actual execution times using asynchronous and synchronous modes compared with ideal linear scaling.

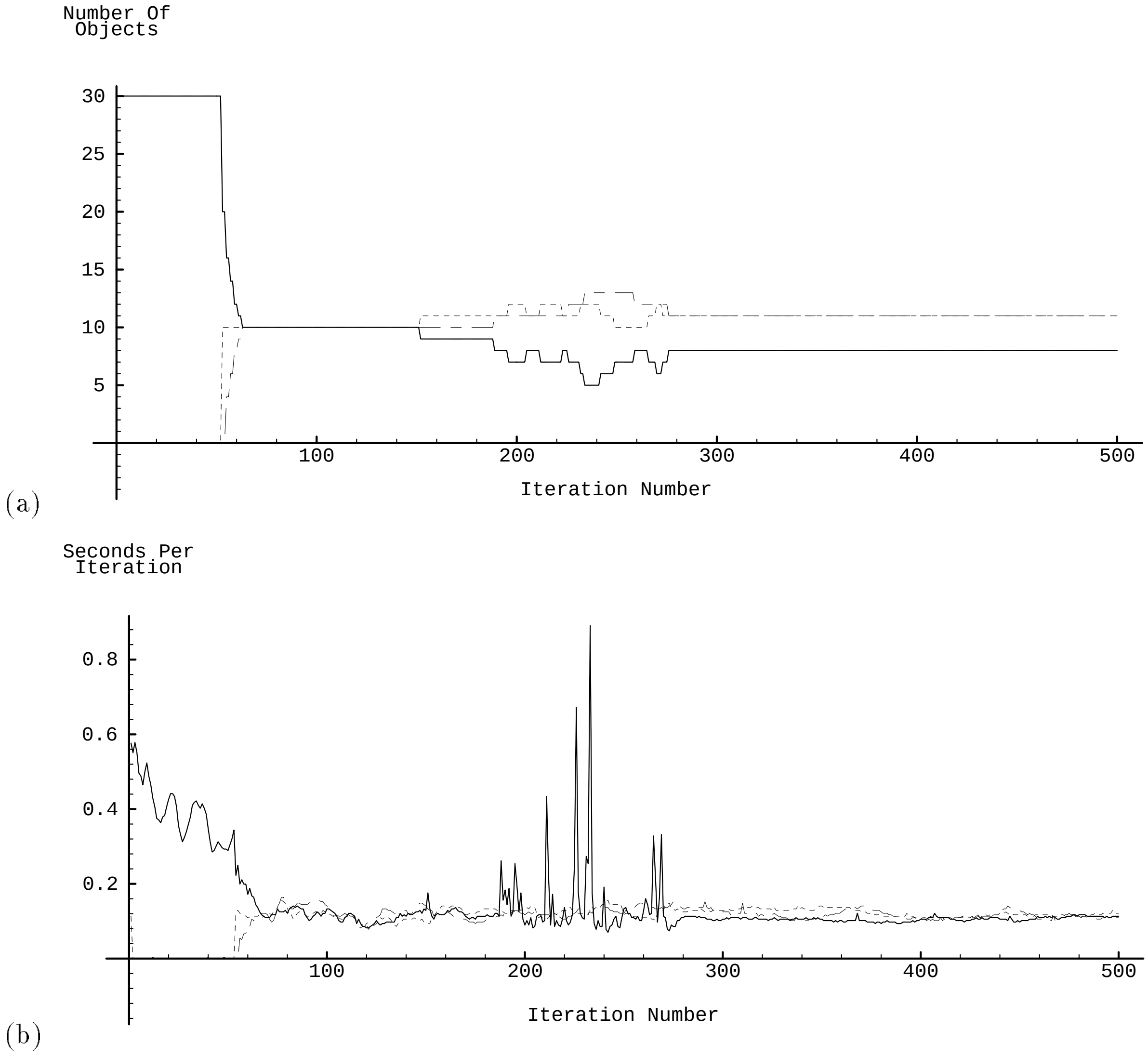


FIGURE 3. Load leveling and task migration: (a) number of objects being simulated by each processor, (b) execution time per simulated time step for each processor

that severely limits the number of machines we can effectively employ. With the addition of broadcast socket code we expect a sharp improvement in performance for simulations with larger numbers of processors.

6.2 Load Leveling and Task Migration

One of the most important characteristics of the system is its ability to adapt to variable system loading. Such variable loading can occur because of user input (e.g., adding a new object), or collisions, or events external to the system (e.g., other users running jobs on the same processor). Whatever the cause, the system must quickly adapt its load distribution in order to maintain the condition of bounded asynchrony.

Figure 3 shows a case of load distribution of 30 like objects among three computer workstations. In this example each workstation is responsible for a physical simulation process and a user interface / graphics process. Figure 3(a) shows the number of objects being handled by each of the three workstations, and Figure 3(b) shows the time each workstation requires to simulate one time step for all of its objects.

This example simulation starts by adding all 30 objects to one processors list and runs for 500 timesteps. In this example the resource allocation code is not activated until the 50th timestep; when it is engaged the three processors quickly (within five to ten iterations) equalize their workloads using the allocation scheme described above.

Sometime around the 150^{th} iteration a command to spawn a new, outside, process was typed into a shell on the processor represented by the solid line. By the 250^{th} iteration, the new process was up and running. As can be seen, the resource allocation scheme quickly adapted to the changes in actual workstation processor load. While the system was paging and spawning the new process, the processor affected acted independently to adjust the three workloads accordingly for the remainder of the simulation. The allocation scheme works similarly well in a variety of other configurations.

7 Summary

We have presented a system for distributed interactive simulation of complex, multi-body situations using either rigid or non-rigid dynamics. The system's efficiency has been demonstrated to scale nicely as a function of the number of processors, up to five processors, the maximum number tested. The system makes use of a novel bounded asynchronous operation mode allowing it to more fully utilize processor and network resources. It maintains this efficiency by allocating computational resources among networked workstations using a simple, efficient "market-based" strategy, thus avoiding problems of central control.

Acknowledgements. We thank one anonymous reviewer for many pages of helpful comments, Digital Equipment Corporation for its generous equipment grant, and the help and support of our collaborators, Thad Starner, Bradley Horowitz, Irfan Essa and Stan Sclaroff.

8 References

[1] A. Barr. Superquadrics and Angle-Preserving Transforms. *IEEE Computer Graphics and Applications*, 1(1):11–23, 1981.

[2] K. Bathe. *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, 1982.

[3] S. Becker and Y.L. Cun. Improving Convergence With Second Order Methods, *Proc. of the Connectionist Models Summer School, CMU, Pittsburgh*, 29-37.

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[5] B. Blau, C. E. Hughes, M. J. Moshell, C. Lisle. Networked Virtual Environments *Computer Graphics*, 26(2):157–160, 1992.

[6] I. Essa., S. Sclaroff, and A. Pentland. A Unified Approach for Physical and Geometric Modeling for Graphics and Animation. In *Proceedings of the European Computer Graphics Conference*, Cambridge, England, 1992.

[7] B. A. Huberman(Ed.). *The Ecology of Computation.* 1988, Amsterdam: North-Holland.

[8] Yoshino Izui and Alex Pentland. Analysis of neural networks with redundancy. *Neural Computation*, 2:225–238, 1990.

[9] T. W. Malone, R. E. Fikes, K. R. Grant and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In B. A. Huberman(Ed.), *The Ecology of Computation* (pp.177-205). Amsterdam: North-Holland.

[10] M. S. Miller and K. E. Drexler. Markets and computation: agoric open systems. In B. A. Huberman(Ed.), *The Ecology of Computation* (pp.133-176). Amsterdam: North-Holland.

[11] A. P. Pentland and J. R. Williams. Good Vibrations : Modal Dynamics for Graphics and Animation. *ACM SIGGRAPH Conference Proceedings*, 23(4):215–222, 1989.

[12] A. Pentland, I. Essa, M. Friedmann, B. Horowitz, and S. Sclaroff. The thingworld modeling system: virtual sculpting by modal forces. *Computer Graphics*, 24(2):143–144, 1990.

[13] A. Pentland and S. Sclaroff Closed form solutions for physically based shape modeling and recovery. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 13(7):715–729, July 1991.

[14] A. Pope. The SIMNET Network and Protocols. BBN Report No. 7102, BBN, BBN Systems and Technologies, Cambridge, MA July 1989.

[15] S. Sclaroff, and A. Pentland. Generalized implicit functions for computer graphics. *ACM Computer Graphics*, 25(4):247–250, 1991.

[16] D. Terzopoulos and D. Metaxas. Dynamic Models with Local and Global Deformations: Deformable Superquadrics. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 13(7):703–714, July 1991.

[17] B. von Herzen, A. Barr, and H. Zatz. Geometric collisions for time-dependent parametric surfaces. *Computer Graphics*, 24(4):39–48, 1990.

[18] M. J. Zyda, D. R. Pratt, J. G. Monahan, K. P. Wilson NPSNET: Constructing a 3D Virtual World *Computer Graphics*, 26(2):147–156, 1992.