Parallel Discrete Event N-body Dynamics

Matthew Holly and Carl Tropper
School of Computer Science
McGill University
Montreal, Canada
matthew.holly@mail.mcgill.ca, carl@cs.mcgill.ca

***Abstract*—Numerical simulation of gravitational N-body systems is an important tool for studying the dynamic behaviour of stellar systems, and in some cases is the only option available given the extremely large time scales involved. The direct summation approach, which evaluates the force between each pair of particles at each time step, produces the most accurate results. However despite many algorithmic advances this method remains a computationally challenging problem owing to its $O(N^2)$ scaling characteristics. The desire to model increasingly larger systems has spurred the adoption of parallel computation techniques, but unfortunately many of the strategies used to accelerate sequential direct N-body simulations hinder their efficient parallelization. This paper investigates the use of parallel discrete event simulation as an alternative to the usual iterative time-stepping approach. By decomposing typical operations into finer-grained events, it is shown that there exists considerable potential for exploiting the model's inherent concurrency. In addition, it is demonstrated how certain optimizations that are normally difficult to parallelize are incorporated naturally into the parallel discrete event paradigm.**

Keywords-parallel, distributed, discrete event, numerical simulation, n-body dynamics.

I. Introduction

The simulation of physical systems, often modelled by partial differential equations covering multiple spatial and temporal scales, has traditionally been one of the main driving forces behind research in the field of high-performance computing. Larger and more complex models place ever-increasing demands on computational resources, and in many cases harnessing the potential of parallel computers is the only way to achieve significant performance improvements.

Most algorithms for simulating physical systems employ a time-stepping approach. The model is represented by variables and data structures at some moment in simulation time t. In order to integrate the state forward, the simulation clock is advanced by some small amount Δt, and the new state is computed to represent the model at time $t+\Delta t$. The process is thereafter repeated until the simulation clock reaches some maximum value or some other convergence criterion has been met. The parallel implementation of time-stepping algorithms is relatively straightforward. Each CPU is allocated a portion of the model, and each integrates its portion of the global state forward from time t to time $t+\Delta t$, whereupon all processes synchronize and exchange updated state information.

Parallel and distributed discrete event simulation (PDES) is an alternative to the more common time-stepping or time-driven approaches, and has successfully been used to accelerate the execution of spatially discretized physical models [24]. Domain decomposition is performed as before, however afterwards the processes advance their subset of the state variables independently at their own rate, communicating asynchronously via time-stamped messages.

Unlike many time-stepping schemes which force a costly global synchronization after every major integration step, PDES synchronization strategies fall into two broad categories: conservative or optimistic. Conservative mechanisms avoid processing any messages beyond a certain simulation time until it can be ascertained that no message with a lower time stamp can possibly arrive [5],[21]. In other words, conservative methods avoid scenarios where an incoming message could arrive in the logical past, which would indicate a causality violation. Optimistic PDES algorithms, on the other hand, do not wait until it is safe to process a message, but do provide mechanisms to detect and recover from causality violations should they occur [14]. By speculatively executing messages and avoiding the inefficiencies associated with global synchronization, optimistic strategies are ideal for extracting a model's inherent parallelism.

In addition to conservative and optimistic synchronization strategies, it is possible to create application-specific protocols when necessary. A given application may present a particular set of requirements, or may be important enough to merit such customized treatment. We argue that the gravitational N-body problem is just such a problem, and present a parallel event-driven synchronization algorithm that exploits additional concurrency not accessible to either conservative or optimistic methods.

A. *Paper outline*

This paper is structured as follows. Section II discusses the time-stepping and event-driven simulation methods, with a focus on highlighting the differences between the two strategies. Section III describes the N-body problem, including

a description of the integration scheme employed to advance particle data. This section also details the impediments to efficiently porting optimizations useful for sequential simulations to parallel computers. Section IV builds on the previous two sections, describing a parallel discrete event implementation of an N-body solver. Section V provides an analysis of the performance of the discrete event algorithm. Section VI covers future research and possible extensions of the work presented herein, while section VII concludes.

II. Discrete simulation methods

Simulation methods that discretize time are distinguishable by the manner in which they order and control the flow of simulation time. The most common strategies for advancing simulation time are normally either of the time-stepped or event-driven variety.

A. Time-stepped execution

In the time-stepped approach simulation time is advanced in constant-sized steps, with state variables being updated as the system progresses through the sequence of steps. The choice of step size is crucial to both the performance and accuracy of the time-stepping method. Choosing a time step that minimizes computational costs while still managing to capture essential simulation characteristics often requires care and experience.

Time-stepping schemes lend themselves naturally to parallel implementation. Each processor is assigned a subset of the state space and is responsible for updating only that subset. Each CPU is then free to advance its own *local* simulation time and update its own segment of the global state space. The simulation proceeds in a lock-step pattern, with all processors required to block at the end of each step in order to synchronize newly updated state data.

For any given time step it's possible that some processors will have less work to do than others and hence will spend a relatively larger amount of time idling at the global synchronization barrier, unable to continue until the others catch up. The pathological case has only a single heavily-loaded CPU active during a given time step, while all others remain idle. Proper load balancing, arising from the initial partitioning of the state space and potentially dynamically adjusted over the course of the simulation, is crucial to maximizing the performance of the time-stepping schemes.

B. Event-driven execution

The trial-and-error process of choosing the right time step is problematic. Rather than updating the state of the system at fixed intervals, the event-driven approach targets pertinent state changes that take place at instantaneous moments in simulation time. Events scheduled at specific times are processed by invoking application-defined event handling functions.

By focusing on interesting moments over the course of the simulation, event-driven schemes have the potential to be more efficient that their time-stepping counterparts. In particular, when the state trajectory is relatively stable except for a few key irregularly dispersed moments, the event-driven method does not waste time recomputing state information when nothing much is going on.

Between the time-stepping and event-driven paradigms, the latter is traditionally seen as the more complex to parallelize. The first difficulty stems from the absence of a fixed time step known to all processors, such that at any given moment in wall clock time one should expect a disparity amongst the simulation times of each processor. The processors advance asynchronously and it becomes more difficult to know when it is safe to advance local state, as receiving a time-stamped message in one's logical past would indicate that a causality violation has occurred.

III. The N-body Problem

This section provides the background information on the N-body problem necessary to later introduce the parallel discrete event implementation of an N-body solver.

A. Overview

Despite a long history, N-body simulation remains a computationally challenging problem. The general N-body problem refers to the study of the dynamics of N interacting bodies, in particular the motion of particles under the influence of their mutual gravitational attraction. N-body simulations have been used to study a wide range of astrophysical models, from the dynamics of galaxies and globular clusters containing on the order of 10^6 stars to the relatively small scale of planetary systems containing dozens of bodies [1],[4].

An N-body system consists of N particles, represented as infinitesimal point masses. The total force acting on a particle is the sum of its pairwise interactions with all other particles.

$$\boldsymbol{F} \equiv m_i a_i = m_i G \sum_{j=1, j \neq i}^{N} m_j \frac{\boldsymbol{x_i} - \boldsymbol{x_j}}{\left| \boldsymbol{x_i} - \boldsymbol{x_j} \right|^3} \qquad (1)$$

B. Numerical Method

The dynamic evolution of particles in the system is driven by their time-varying mutual gravitational field. Direct summation approaches involve the straightforward application of Newton's law (1). The force on each particle is calculated by accumulating the contributions of all other particles in the system, for a total of N(N-1)/2 partial force calculations. By explicitly computing the inter-particle forces for all pairs of particles in the system, this approach scales as $O(N^2)$ and is thus computationally very demanding for systems involving large numbers of particles. Direct evaluation methods are typically employed for star clusters containing on the order of

10^4 stars [18], whereas the use of special-purpose hardware has enabled systems with 10^5 stars to be studied using direct evaluation techniques [17].

A variety of methods have been developed to alleviate the burdensome $O(N^2)$ scaling properties of the brute force scheme. The basic idea is to reduce the number of direct particle-particle force evaluations by using approximation techniques. For example, the Barnes-Hut tree method works by partitioning the simulation volume into an hierarchical tree of axis-aligned sub-volumes or cells [3]. The leaf nodes of the tree contain the particles, and the force on each particle is calculated by walking the tree. The hierarchical nature of the tree allows for the gravitational potential due to entire groups of particles to be approximated by low-order multipole expansion [3].

Despite the advantageous scaling characteristics of the Barnes-Hut scheme and its ability to handle significantly more particles, in many cases the direct calculation method remains preferable owing to its superior accuracy, algorithmic simplicity, and the fact that no *a priori* assumptions need to be made regarding the system being simulated (e.g. uniform particle distribution) [23]. For these reasons we shall restrict ourselves to direct summation methods for the remainder of this paper.

C. Individual Time Steps

In the physical systems modelled by N-body simulations it is typical to observe various phenomena operating at vastly different time scales [1]. Correspondingly, the optimal choice of time step to best capture particle interactions can vary greatly over the course of each particle's trajectory.

Individual time steps (ITS) are a significant improvement over the constant time step method, offering several orders of magnitude increase in performance as well as improved accuracy [19]. The idea is to assign each particle P_i a suitable time step Δt_i, reserving small time steps for only those particles that require them while allowing others to take larger steps whenever possible. In essence, stepping forward on a per-particle basis is a response to a commonly observed characteristic of N-body simulations wherein few particles require frequent and smaller time steps in order to preserve the numerical stability of their trajectory [1]. Most direct N-body codes incorporate some variation of the individual time step approach, however the use of ITS often precludes an efficient parallel implementation [25].

D. The Hermite Scheme

In this section we describe the widely-used 4th order Hermite integration scheme with individual time steps [18]. Each particle P_i has mass m_i, position $\mathbf{x}_i$, velocity $\mathbf{v}_i$, acceleration $\mathbf{a}_i$, as well as the first time derivative of the acceleration $\mathbf{k}_i$ (known as the “jerk”). In the case of individual time steps, each particle also has its own time t_i and step Δt_i. A particle's time reflects the last time it was updated, and its update time $ut_i = t_i + \Delta t_i$ is the next time at which the forces acting on the particle will be calculated.

The time-stepping algorithm proceeds iteratively, as follows:

1) Each particle's initial time step is calculated using (2). As per Makino and Aarseth [18], a scaling constant with value of $\eta_s \sim 0.01$ is typically sufficient.

$$\Delta t_i = \eta_s \left| \frac{\boldsymbol{a}_i}{\boldsymbol{k}_i} \right| \tag{2}$$

2) Select particle P_i with the smallest update time $ut_i = t_i + \Delta t_i$ and set the global simulation clock to this time.

3) Predict the positions and velocities of all particles to the update time ut_i. This prediction is done via extrapolation using the values **x**, **v**, **a** and **k**, as per (3) and (4). Note that in general for individual time step methods each particle will have its own local time and hence the value of $dt_j = ut_i - t_j$ will differ.

$$\boldsymbol{x}_{pj} = \boldsymbol{x}_j + \boldsymbol{v}_j dt_j + \frac{1}{2}\boldsymbol{a}_j dt_j^{\,2} + \frac{1}{6}\boldsymbol{k}_j dt_j^{\,3} \tag{3}$$

$$\boldsymbol{v}_{pj} = \boldsymbol{v}_j + \boldsymbol{a}_j dt_j + \frac{1}{2}\boldsymbol{k}_j dt_j^{\,2} \tag{4}$$

4) Calculate the acceleration $\mathbf{a_p}$ and jerk $\mathbf{k_p}$ using direct summation as per (6) and (7). This is the most compute-intensive part of any N-body kernel.

$$\begin{aligned} \boldsymbol{r}_{ij} &= \boldsymbol{x}_{pj} - \boldsymbol{x}_{pi} \\ \boldsymbol{v}_{ij} &= \boldsymbol{v}_{pj} - \boldsymbol{v}_{pi} \end{aligned} \tag{5}$$

$$\boldsymbol{a}_p = \sum_j G m_j \frac{\boldsymbol{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} \tag{6}$$

$$\boldsymbol{k}_p = \sum_j G m_j \frac{\boldsymbol{v}_{ij}}{(r_{ij}^2 + \epsilon^2)^{3/2}} + \frac{3(\boldsymbol{v}_{ij} \cdot \boldsymbol{r}_{ij})\boldsymbol{r}_{ij}}{(r_{ij}^2 + \epsilon^2)^{5/2}} \tag{7}$$

5) Using a Hermite interpolation based on the acceleration and jerk, calculate the higher order derivatives of the acceleration with (8) and (9).

$$\ddot{\boldsymbol{a}}_i = \frac{-6(\boldsymbol{a}_i - \boldsymbol{a}_p) - dt(4\boldsymbol{k}_i + 2\boldsymbol{k}_p)}{dt^2} \tag{8}$$

$$\ddot{\boldsymbol{k}}_i = \frac{12(\boldsymbol{a}_i - \boldsymbol{a}_p) + 6\,dt(\boldsymbol{k}_i + \boldsymbol{k}_p)}{dt^3} \tag{9}$$

6) The predicted position and velocity from step 3) are then corrected to higher order using (10) and (11).

$$\boldsymbol{x}_i = \boldsymbol{x}_p + \frac{\ddot{\boldsymbol{a}}_i}{24} dt^4 + \frac{\ddot{\boldsymbol{k}}_i}{120} dt^5 \tag{10}$$

$$\boldsymbol{v_i} = \boldsymbol{v_p} + \frac{dt^3}{6}\ddot{\boldsymbol{a}}_i + \frac{dt^4}{24}\ddot{\boldsymbol{k}}_i \tag{11}$$

7) At this point particle P_i is fully updated. We set $t_i = t_i + \Delta t_i$ and then choose a new time step Δt_i for P_i's next update according to the standard Aarseth formula [18] given by (12).

$$\Delta t_i = \sqrt{\eta \frac{|\boldsymbol{a}_p||\ddot{\boldsymbol{a}}_i| + |\boldsymbol{k}_p{}^2|}{|\boldsymbol{k}_p||\ddot{\boldsymbol{k}}_i| + |\ddot{\boldsymbol{a}}_i{}^2|}} \tag{12}$$

8) If the global simulation clock hasn't reached the simulation end time, go back to step 2) and repeat.

E. Parallelization

Many important astrophysical problems, for example the simulation of globular clusters, involve large numbers of particles (on the order of 10^6) and require a highly accurate integration method. The direct force computation scheme, owing to its $O(N^2)$ scaling characteristics, is not normally employed for such extremely large systems. With the use of special purpose hardware such as the GRAPE-6, simulating systems on the order of 10^5 particles has become possible [17], [10]. However as with most custom hardware configurations, accessibility remains limited and thus there has been considerable effort put towards parallelizing N-body kernels for execution on general purpose parallel computers [25], [9].

There are two common strategies employed when partitioning N-body systems amongst multiple processors. In the distributed data method, each processor stores particle data for only a subset of the overall system. To compute the forces acting on particles in its local subset, the processor sends a copy to the other nodes, either via broadcast or with an overlaid logical topology as in the systolic ring algorithm [9]. Partial force contributions are computed by the receiving node, after which particles are returned to the source node for the correction phase of the integrator. In the case of the systolic ring method, partial forces are aggregated as the data cycles from neighbour to neighbour.

By contrast, in the replicated data strategy each processor has a complete copy of all particles in the system and is responsible for updating a subset of these particles. This has the advantage that each node can independently compute the forces on its local group. The downside is the higher memory consumption and the costly global synchronization at the end of each update step. For large enough values of N, the performance of the systolic ring and replicated data algorithms is comparable [9]. We thus test only against the replicated data strategy in this paper.

Unfortunately, many of the methods traditionally used to accelerate direct N-body codes appear to hinder the effective parallelization of the algorithms. In the case of individual time steps, it's possible for too few particles to be active on a given update step. In a distributed context, this could result in some processors having little or no particles to update, forcing them to wait idly until the next global synchronization.

Most parallel implementations of the Hermite scheme use some form of hierarchical or block time step [25],[9]. In this approach, time steps are constrained to certain blocks of allowable step sizes, resulting in larger groups of particles being advanced concurrently during each step.

The block time step approach is a compromise between the accuracy provided by individual time steps and the simple parallel implementation afforded by constant time steps. For situations where even a small reduction in accuracy is unacceptable, we seek a method that can retain the original benefits of the individual time step strategy while also being naturally extensible to parallel implementation.

IV. Parallel discrete event N-body simulation

A. Discrete event formulation

Although normally seen as an iterative time-stepping algorithm, numerical N-body simulations exhibit enough traits of typical discrete event problems so as to render them amenable to treatment as such. In this section we detail discrete event formulations of the N-body problem for both constant-sized and variable per-particle time steps. In the descriptions that follow, we focus exclusively on systems where all processors have a complete copy of the system's particles and are responsible for advancing a subset of the overall particles. As the processors advance their state asynchronously local copies of remote particle data may become out of date; it is the responsibility of the distributed control mechanism to ensure that causality violations are avoided.

The following two sections introduce parallel discrete event N-body algorithms, firstly for the constant time step (CTS) case and thereafter extended to handle individual time steps (ITS).

B. Discrete event CTS

The obvious and natural place to start is to treat each iteration of the integration loop as a distinct STEP event, with each such event scheduling its successor. Hence an update event at time t_1 will schedule the next update event at time $t_2=t_1+\Delta t_1$, which in turn will schedule an event at $t_3=t_2+\Delta t_2$, and so on. In the case of constant time steps this is simplified to $t_{n+1}=t_n+\Delta t$ since the size of the time step is invariable.

When considering a parallelized version of this strategy, we must also take into account the updates from neighbouring processors, thus local step events are only scheduled once all outstanding updates have arrived from the other processors. This ensures that no local step is taken before first having an up-to-date copy of the global state, without resorting to a costly global synchronization barrier. The rest of this section

```
N  := set of all particles
L  := set of local particles
P  := set of all processors

# Event handler for step events in CTS mode.
procedure event_step_cts(Event e)
{
   # Get the timestamp of the event.
   t = timestamp(e)

   # Predict position and velocity of all particles to the update
   # time t.
   predict(N, t)

   # Calculate acceleration and jerk of all local particles
   # at time t.
   calculate_force(N, L, t)

   # Correct predicted quantities for all local particles.
   correct(L, t)

   # Pack updated position, velocity, acceleration and jerk into a
   # message.
   msg = create_message(L, t)

   # Send the message to all other nodes.
   broadcast_message(msg, P)
}
```

Figure 1. Step event for constant time steps.

describes the algorithm in more detail.

The various phases of the Hermite algorithm are clearly evident in the pseudocode for the step event (Fig. 1). Operating on the local copy of the particle system, we first predict the positions and velocities of all particles to the update time *t*. We then compute the acceleration and time derivative of the acceleration for all local particles, using the predicted particle state. As before, this is the most computationally intensive part of the overall simulation. Again while only operating on local particles, we compute the higher order derivatives of the acceleration and jerk, and use them to apply the correction to the position and velocities that were predicted earlier on. At this point, the local particles have been updated.

The next portion of the step event is a divergence from the time-stepping algorithm. Recall that in the parallel discrete event simulation paradigm, processors communicate by sending time-stamped messages that have the dual effect of updating local copies of state data as well as providing information essential for local synchronization. The updated portion of each local particle, namely its position, velocity, acceleration and jerk, are packed into a message structure. The message is stamped with the current local virtual time, which corresponds to the new time at which the particle information is valid.

The reception of an incoming message containing updated particle data triggers a second type of event: the remote update (RUP) event. Each processor sends its updated state information to all other nodes, which implies that each should expect to receive P-1 such RUP messages (where P is the number of processors). Only when all outstanding messages have been processed can we proceed to the next step event, and correspondingly we refrain from scheduling the next local step event until all particles have been updated.

Fig. 2 shows the pseudocode for the RUP event handler.

```
N  := set of all particles
L  := set of local particles
P  := set of all processors

# Event handler for step events in CTS mode.
procedure event_rup_cts(Event e)
{
   # Get the timestamp of the RUP event.
   t = timestamp(e)

   # Update counter that tracks the number of outstanding updates.
   remaining_updates = remaining_updates - 1

   # Apply remote update to synchronize local data.
   apply_rup(e, R)

   # Test if there are still outstanding remote updates for
   # this step.
   if (remaining_updates == 0)
   {
      # Reset the update count (expect one update from each
      # other node)
      remaining_updates = |P| - 1

      # Schedule the next event
      dt = time_of_next_step(L, t)
      schedule_step(dt)
   }
}
```

Figure 2. Remote update event for constant time steps.

The function apply_rup(), which performs the actual processing of the contents of the remote message, is straightforward (Fig. 3). For each remote particle in the message, we update the position, velocity, acceleration and jerk fields in our local copy. Once this is done, our local copy of the sending processor's particles has been synchronized.

We have shown so far that the discrete event method can be used to emulate the more common time-stepping approach for N-body simulation. Since all required updates are sent and eventually arrive at the destination nodes, there is no risk of deadlock. Since no new STEP events are scheduled until all remote updates have been processed, particles are guaranteed to be advanced in the correct order and no causality violations can occur.

Unlike the time-stepping approach, no costly global synchronization is required at the end of each step. This is beneficial because the performance of barrier synchronization methods can vary widely due to hardware configuration and software implementation. Nevertheless, the computational workload must be evenly balanced amongst the processors in

```
N  := set of all particles
U  := local update list
R  := set of remote particles contained in message

# Apply remote update to synchronize local data.
procedure apply_rup(Event e, Set R)
{
   # Get the timestamp of the RUP event.
   t = timestamp(e)

   # Apply the remote update
   for r in R:
   {
      # Find local copy of this remotely-owned particle
      p = find_particle(N, r->id)

      # Copy the updated component vectors
      p->position     = r->position
      p->velocity     = r->velocity
      p->acceleration = r->acceleration
      p->jerk         = r->jerk
   }
}
```

Figure 3. Applying a RUP event to update local image of remotely-owned particles.

order to avoid situations where some CPUs wait idly for the remote updates to arrive from the slower or more heavily loaded processors. For direct evaluation methods this is achieved simply by allocating an equal amount of particles to each processor, however this task is made more difficult in the context of heterogeneous clusters where the relative clock speeds of CPUs may vary, meaning that slower processors should be allocated less work than faster processors. We therefore seek to improve on this scheme by permitting processors to do useful work while waiting for incoming messages, effectively taking advantage of additional concurrency inherent in the N-body problem that's not available when constant time steps are used.

C. Discrete event ITS

This section describes a discrete event implementation of the Hermite scheme using individual per-particle time steps (ITS). We seek an algorithm that alleviates some of the difficulties in parallelizing ITS that were discussed previously in section III.

The primary impediment to the efficient parallelization of the time-stepping kernel when ITS is used is that some processors may remain idle while others advance their local particles. The particles on the smallest time step are selected for updating at the start of an iteration, and since all processors need to synchronize at the end of each iteration in order to exchange updated state data, processors with no particles on the smallest step have nothing to do. The problem is exacerbated as the range of allowable time steps is expanded, as we should expect more variation amongst the time steps of the system's particles.

The question then is whether the idle processors can manage to do useful work while waiting for outstanding updates from other processors. Although the particles on the smallest time step must be advanced prior to any others, we observe that some of the information necessary to advance particles on the *next smallest* time step is already available (in general, the results of all smaller time steps are missing). For example, if the smallest time step is at time t_1 and the next smallest steps are t_2, t_3 and t_4 respectively, then we can compute the acceleration of particles on step t_2 due to the particles on steps t_3 and t_4, but not due to those particles on step t_1. Upon completion of step t_1, the updated state information is disseminated amongst the nodes and can be used to calculate the remaining force contributions for the particles on step t_2. The crux of the strategy is thus to *partially advance the particles on future time steps, completing these steps later as the required missing data arrives*. We note that such an approach is not possible if processors are forced to synchronize at the end of each step prior to commencing the next step, as is the case with the time-stepping approach.

For a given particle P_i with update time $ut_i=t_i+\Delta t_i$, we say that all particles with update times less than ut_i are *unsafe*. In other words, in order to be factored into the computation of P_i's acceleration at time ut_i, unsafe particles must be advanced to a time such that their next update time is equal to or greater than ut_i. Naturally, without all the required force contributions at ut_i, P_i cannot be advanced and so we say it is *blocked*. Unsafe particles can be either local or remote particles. Safe particles, on the other hand, are all particles with update times equal to or greater than ut_i. Note that with the ITS approach each particle has its own update time, and so the set of particles considered as safe or unsafe will in general be different from particle to particle. In the proper sequential ordering of particle updates, P_i will be updated before (or at the same time as) the particles it considers as safe. The crucial observation here is that we can calculate all force contributions on P_i at time ut_i due to safe particles, but we can only advance P_i when there are no unsafe particles relative to P_i.

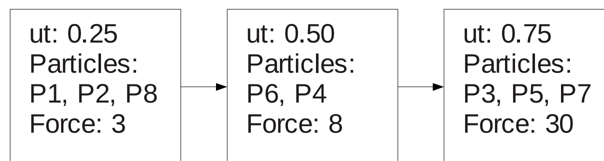

Figure 4. Update list sorted by increasing update time (ut).

To implement this strategy, each processor maintains an *update list* sorted by increasing update time. Each item in the list contains a list of local particles to update (the *update group*), the time at which these particles are to be updated, and a counter to track the number of force computations necessary to completely advance the particles on the update step. Fig. 4 shows an example of a processor's local update list. The item at the head of the list represents the local particles to be updated at time t=0.25. There are three particles in the update group, and a total of three force calculations are needed to completely advance the group. Thus we can deduce that there is exactly one unsafe particle residing on some other processor that is blocking this update.

```
N  := set of all particles
U  := local update list

# Event handler for step events in ITS mode.
procedure event_step_its(Event e)
{
   # Get the timestamp of the event.
   t = timestamp(e)

   # Find the update object associated with this step.
   u = find_update(U, t)

   # Predict all particles that can contribute.
   safe_predict(N, t)

   # Compute partial acceleration and jerk contributions
   # from safe particles only.
   safe_force(N, u)

   # Check if all required force contributions were calculated.
   if (force_remaining(u) == 0)
   {
      # Complete this step.
      complete_updates(U)

      # Schedule future steps.
      schedule_steps(N, U)
   }
}
```

Figure 5. Step event for individual time steps.

```
# Event handler for step events in ITS mode.
procedure event_rup_its(Event e)
{
   # Get the timestamp of the RUP event.
   t = timestamp(e)

   # Apply remote update to synchronize local data.
   apply_rup(e, R)

   # Propagate RUP forward for in-progress updates
   propagate_rup(t)

   # Scan update list for any completed updates
   complete_updates(U)

   # Schedule future steps.
   schedule_steps(N, U)
}
```

Figure 6. Remote update event for individual time steps.

The second update in the update list (at time t=0.50) contains two particles. Note that to complete this group eight force contributions are required, which implies that this group depends on four particles: the same unsafe remote particle that blocks the t=0.25 update, and the three local particles in the t=0.25 group. The final element in the list contains three local particles and has 30 outstanding force contributions, hence it depends not only on the five local particles in the t=0.25 and t=0.50 groups, but also on 5 other unspecified remote particles. Naturally those groups furthest in the future will have the most dependencies, since more particles will have to be updated prior to the group being advanced.

The parallel discrete event implementation of this scheme uses the same two events as in the CTS case, namely a STEP event to schedule local particle updates and a remote update (RUP) event to incorporate incoming updates from other processors. Fig. 5 shows the general outline of a STEP event for the individual time step method. We first search the update list for the group corresponding to the time stamp of the STEP event. (The update group is guaranteed to exist as it is created and inserted into the update list when the STEP event is scheduled.) We then proceed to predict the position and velocity of all *safe particles* only, and then calculate the force contributions of these particles on those in the update group. If after this operation all force contributions have been accounted for, then there were no unsafe particles to block this step, and we can advance the particles in the update group and schedule future steps. If however there are additional force calculations to be performed, then we must wait for the unsafe particles blocking this group to be advanced via RUP events and any subsequent unblocking of local particles upon which this group depends.

```
N  := set of all particles
U  := local update list
R  := set of remote particles contained in message

# Propagate RUP forward for in-progress updates
procedure propagate_rup(time t)
{
   # Loop over the local update list.
   for u in U:
   {
      # The update time of the in-progress update
      ut = update_time(u)

      # Only contribute to partially done future updates
      if (started(u) AND t < ut)
      {
         # Predict the remote particles to ut.
         predict(R, ut)

         # Predict local particles in update set to ut.
         predict(u, ut)

         # Calculate acceleration and jerk components
         # due to remote particles.
         for r in R:
         {
            # Calculate force contribution of r on particles in u.
            calculate_force(r, u)
         }
      }
   }
}
```

Figure 7. Propagation of a remote update through the local update list.

```
N  := set of all particles
U  := local update list
P  := set of all LPs

# Scan update list for any completed updates.
procedure complete_updates(Set U)
{
   for u in U:
   {
      # Complete u if there are no more force contributions to
calculate.
      if (force_remaining(u) == 0)
      {
         # Correct predicted positions and velocities.
         correct(u)

         # Compute next time steps for all particles in u.
         update_time_steps(u)

         # Merge completed update into other ongoing updates.
         merge_update(U, u)

         # Pack updated position, velocity, acceleration
         # and jerk into a message.
         msg = create_message(u, update_time(u))

         # Send the message to all other processors.
         broadcast_message(msg, P)

         # Remove u from the update list.
         remove_from_list(U, u)
      }
   }
}
```

Figure 8. Advancing local particles in update groups that have received all outstanding force contributions.

Remote update events for the ITS scheme (Fig. 6) are likewise slightly more involved than in the CTS case. After first applying the update to synchronize the local image of remotely-owned particles, we then propagate the newly updated state data through the local update list (Fig. 7). This has the effect of adding more force contributions to the in-progress local updates, since the particles updated as part of this RUP event must have been classified as unsafe for any update group with update time after the time stamp of this event.

When there are no remaining force contributions to be calculated on the particles of a local update group, we can complete this update step (Fig. 8). We first apply the correction phase of the Hermite integrator to correct the positions and velocities predicted at the start of the STEP event. Afterwards, we compute a new time step for each of the updated local particles using (12). Care must be taken if the next update time for one of these particles matches another that's already in the local update list. In this situation, there are two options. If no force contributions have been calculated for this update group (i.e. the corresponding STEP event has been

TABLE I. Detailed specifications for Linux clusters

Name	*CPU model*	*Node Configuration*	*Clock speed*	*Memory (per node)*
krylov	AMD Opteron 2376	27 dual-socket, quad core	2.3 GHz	16 GB
alef	Intel Xeon E5410	8 dual socket, quad core	2.33 GHz	16 GB
colosse	Intel Nehalem-EP	x6275 blades (dual-node, dual socket) 7680 cores total	2.8 GHz	24 GB

scheduled but not executed), we simply append the particle to the update group and continue. However if the update group has already been partially advanced, then we must ensure that the same force calculations are applied to the particle before it's added to the group. In other words, the particle must catch up to this group prior to joining it. Merging particles into an update group is a two phase operation. First we compute the safe forces on the newcomers, and then we compute the forces on the update group due to the newcomers. This ensures that all the particles in the update group will have interacted with exactly the same list of particles, which in turn guarantees that no force contributions will be missed later on.

V. Performance of the discrete event method

In this section we compare the performance of the time-stepping and discrete event implementations of the individual time step N-body algorithm.

A. Experimental Setup

Data was generated on three Linux clusters, *krylov*, *alef*, and *colosse*. The krylov cluster is actually a heterogeneous system consisting of 21 quad-core nodes (2.2 GHz) and 27 eight-core nodes (2.3GHz), although we restrict our usage to the 8-core nodes in order to ensure that execution times between runs remain comparable. All computations were performed in double-precision floating-point arithmetic.

B. The Plummer model

We use the well known Plummer model for all experiments (Fig. 9). The Plummer model is a spherically symmetric, centrally concentrated potential model created to fit observational data [4]. We use the system of standard N-body units whereby the total mass of all particles, the radius of volume containing all particles, and the gravitational constant G are all set to unity [12]. We measure wall clock time for the integration of one N-body time unit, referred to as the crossing time, which is proportional to the number of particles in the system.

Initially, the total energy of the Plummer sphere is E=-1/4 [1]. As the simulation progresses, the finite precision of floating-point numerical calculations causes this value to drift; measuring this drift at periodic intervals is a common method of evaluating the accuracy of an N-body integrator. We compute the total energy every 1/8 of a crossing time. The results of the time stepping code are perfectly reproducible, whereas since the discrete event method may process events in a different order from one run to the next (without, of course, introducing any causality violations), the total energy values may vary slightly. We have failed to find any significant difference in energy drift between the two methods, confirming the viability of the discrete event approach and validating the correctness of both codes.

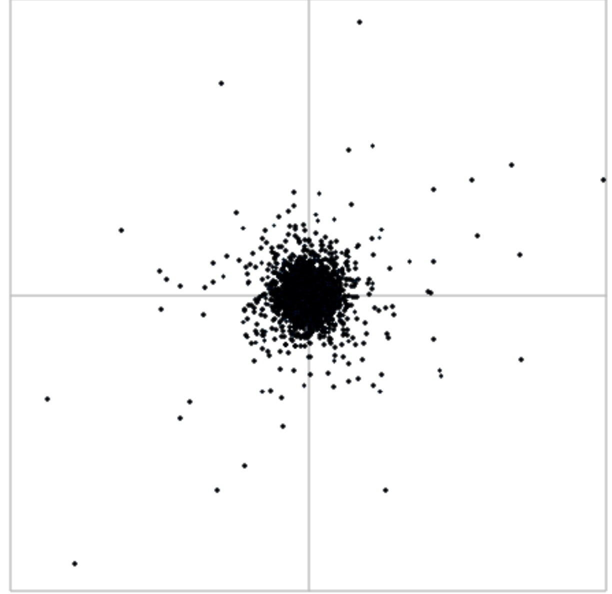

Figure 9. Plummer model (N=16384)

C. Performance analysis

Fig. 10 shows a comparison between the discrete event (DE) and time-stepping (TS) algorithms. The maximum time step was set to t_{max}=1/4 and the minimum time step to t_{min}=1/512. We measured the total execution time on *alef* using 8 and 16 CPUs, for particles systems of size 2^{10} to 2^{14}. The total execution time increases dramatically with the number of particles, illustrating the problematic $O(N^2)$ nature of the direct

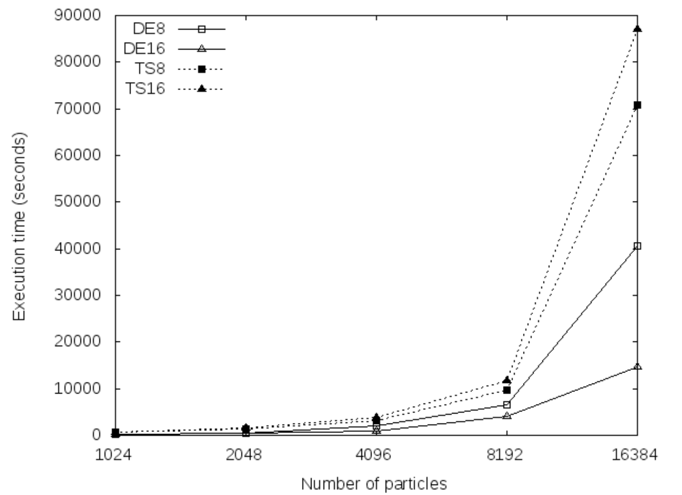

Figure 10. Scaling of the discrete event and time stepping methods on alef.

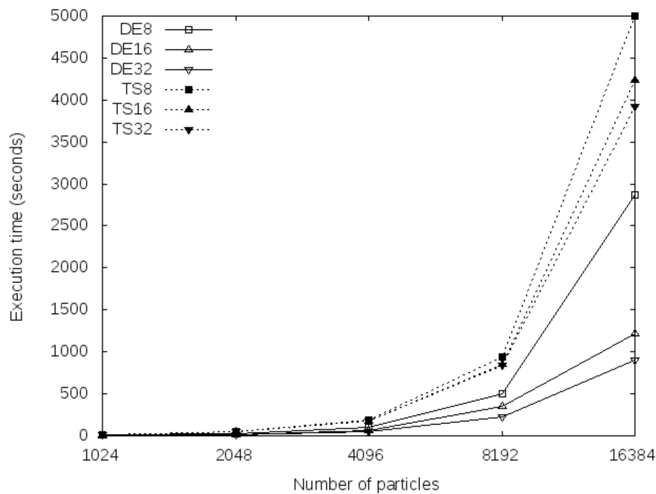

Figure 11. Scaling of the discrete event and time stepping methods on krylov.

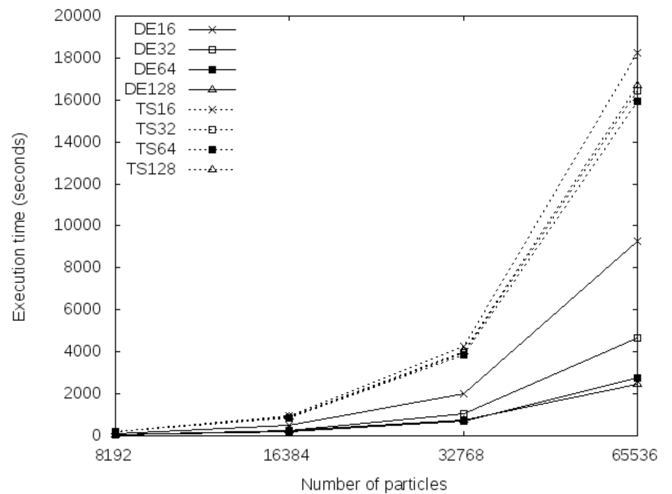

Figure 12. Scaling of the discrete event and time stepping methods on colosse.

evaluation method for N-body problems. The discrete event algorithm (shown by the solid lines in all the graphs that follow, as opposed to the dashed line for time-stepping) scales better as the number of particles increases. It is particularly interesting to note that the performance of the time-stepping algorithm is worse for 16 processors than for 8, implying that synchronization overhead and wasted CPU cycles nullify any performance benefits seen when using the individual time step scheme on a single processor. Additionally, since the nodes of *alef* contain 8 cores, performing a barrier synchronization is clearly more expensive once more than 8 processors are involved, resulting in this surprising performance reversal when 16 processors are used. The discrete event method, on the other hand, is better able to utilize the extra processors, resulting in a 64% reduction for N=16384 when the number of processors is doubled.

Fig. 11 shows the results of a similar run on the *krylov* cluster. A larger minimum time step of $t_{min}=1/128$ was used to demonstrate how both algorithms fare when faced with a smaller range of particle step sizes. We also tested 8, 16 and 32 processor configurations, again restricting ourselves to the 8-core nodes. Both algorithms perform better as more processors are added, however the improvement is more evident in the discrete event case. Perhaps more importantly, the discrete event algorithm outperforms the time-stepping strategy by a significant margin. In particular, even with only 8 processors the discrete event code manages to be almost 27% faster than the time stepping run with 32 processors. It appears that the discrete event method is better able to exploit the additional concurrency not available to methods that rely on global synchronization and lock-step execution.

The next example (Fig. 12) demonstrates how both approaches scale as the number of processors is increased. Data for this run was generated on *colosse*, the biggest of the three Linux clusters. Each data point displayed represents an average over four runs with identical parameters. The dashed lines representing the execution times of the time stepping method are all grouped together; clear evidence that adding additional processors does not yield any performance improvements in this case.

There is much more variety for the solid lines representing the results of the discrete event runs. For N=65536 particles the discrete event code outperforms the time stepping code by approximately 49%, 72%, 83% and 85% when 16, 32, 64 and 128 processors are used, respectively. The processors are kept busier by partially computing particle updates with the information on hand, and do not waste nearly as many cycles idling. The cost of performing a global barrier synchronization increases with the number of processors, which is another factor behind the poor performance of the time stepping code.

Although adding more processors does reduce the total execution time for the discrete event algorithm, we note in Fig. 12 that the is little difference between runs using 64 and 128 processors. In order to determine if larger particle systems are required for performance to continue to scale well with the number of processors, we increased the maximum number of particles to 2^{18}. To reduce the overall simulation time, the

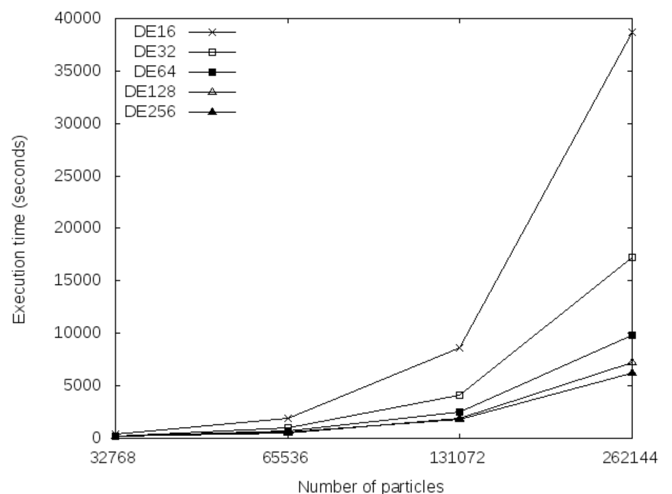

Figure 13. Scaling of the discrete event algorithm for larger particle systems.

minimum time step was increased to t_{min}=1/16. Fig. 13 shows the results for 16 to 256 processors (as before, execution time is the average from four runs). For the largest particle systems tested, we observe successive speed-ups of approximately 44%, 43%, 27% and 14% each time the number of processors is doubled, from the initial 16 to a total 256 CPUs.

VI. Future work

We believe the novel application of parallel discrete event simulation techniques presented herein provides an overture to additional applications within the domain of numerical N-body methods. In particular, we plan on extending our method to handle a variation of the Hermite integrator that employs particle-centric neighbourhoods to divide the force acting on a particle into *near* and *far* components, as described by Makino and Aarseth [18]. We plan on further extending our scope to encompass the spatial partitioning trees used in the Barnes-Hut scheme [3].

VII. Conclusion

We have described a discrete event implementation of an N-body solver using individual per-particle time steps. Whereas the use of individual time steps tends to accelerate sequential N-body simulations, in practice is is often an impediment to the efficient parallel implementation of N-body kernels. We have shown that the discrete event implementation using an application-specific control protocol outperforms the time stepping implementation and scales better with the number of processors. In various other application domains such as molecular dynamics and rigid body simulation, event-driven methods have been employed successfully to accelerate sequential simulations. Despite this potential, event-driven schemes are often seen as more difficult to parallelize than traditional time stepping approaches. We believe the work presented herein will lead to simple, efficient parallel discrete event simulations in other these and other domains.

References

[1] S. J. Aarseth, "Gravitational N-body simulations," Cambridge University Press, Cambridge, UK, 2003.

[2] S. J. Aarseth, "From NBODY1 to NBODY6: the growth of an industry," Publications of the Astronomical Society of the Pacific, 1999, Vol. 111, Issue 765, pp. 1333-1346.

[3] J. Barnes, P. Hut, "A hierarchical O(N log N) force-calculation algorithm," Nature, 1986, ISSN: 0028-0836, vol. 324, pp. 446-449.

[4] J. Binney, S. Tremaine, "Galactic Dynamics," Princeton University Press, Princeton, NJ, 1987.

[5] K. M. Chandy, J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications of the ACM, 1981, Vol. 24, No. 11, pp. 198-206.

[6] A. Donev, "Asynchronous Event-Driven Particle Algorithms," SIMULATION, 2009, vol. 85, no. 4, pp. 229-242.

[7] R. Fujimoto, "Parallel and Distributed Simulation Systems," John Wiley & Sons Inc., New York, NY, 1999.

[8] R. Fujimoto, "Parallel Discrete Event Simulation," Communications of the ACM, 1990, Vol. 33, No. 10, pp. 30-53.

[9] A. Gualandris, S. P. Zwart, A. Tirado-Amos, "Performance analysis of direct N-body algorithms for astrophysical simulations on distributed systems," Parallel Computing, 2007, ISSN: 01678191, vol. 33, pp. 159-173.

[10] S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. P. Zwart, P. Berczik, "Performance analysis of direct N-body algorithms on special-purpose supercomputers," New Astronomy, 2007, vol. 12, Issue 5, pp. 357-377.

[11] D. C. Heggie, "The Classical Gravitational N-Body Problem," astro-ph/0503600, 2005.

[12] D. C. Heggie, R. D. Mathieu, "Standardized Units and Time Scales," In P. Hut and S. McMillan, eds, The Use of Supercomputers in Stellar Dynamics, 1986, Springer-Verlag, Berlin, pp. 233.

[13] P. Hontalas, B. Beckman, "Performance of the Colliding Pucks Simulation on the Time Warp Operating System (Part 2: A Detailed Analysis)," In Proceedings of the 1989 Summer Computer Simulation Conference, J. Clema, ed., Society For Computer Simulation, 1989, pp. 91-95.

[14] D. Jefferson, "Virtual Time," ACM Transactions on Programming Languages and Systems, 1985, Vol. 7, No. 3, pp 404-425.

[15] D. G. Korycansky, "Orbital Dynamics for Rigid Bodies," Astrophysics and Space Science, 2004, Vol. 291, No. 1, pp. 57-74.

[16] Z. Leinhardt, D. C. Richardson, T. Quinn, "Direct N-body Simulations of Rubble Pile Collisions," Icarus, 2000, vol. 146, part 1, pp. 133-151.

[17] J. Makino, T. Fukushige, M. Koga, K. Namura, "GRAPE-6: The massively-parallel special-purpose computer for astrophysical particle simulation," Publications of the Astronomical Society of Japan, 2003, ISSN: 0004-6264, vol. 55, part 6, pp. 1163-1188.

[18] J. Makino, S. J. Aarseth, "On a Hermite Integrator with Ahmad-Cohen Scheme for Gravitational Many-Body Problems," Publications of the Astronomical Society of Japan, 1992, 44, pp. 141-151.

[19] J. Makino, "Optimal Order and Time-Step Criterion for Aarseth-Type N-Body Integrators," The Astrophysical Journal, 1991, 369, pp. 200-212.

[20] B. Mirtich, "Timewarp Rigid Body Simulation," In Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques, International Conference on Computer Graphics and Interactive Techniques, 2000, pp. 193-200.

[21] J. Misra, "Distributed Discrete-Event Simulation," ACM Computing Surveys, 1986, Vol. 18, No. 1, pp. 39-65.

[22] D. Richardson, P. Michel, K. J. Walsh, K. W. Flynn, "Numerical simulations of asteroids modelled as gravitational aggregates with cohesion," 2009, Planetary and Space Science, Vol. 57, Issue 2, pp. 183-192.

[23] R. Spurzem, "Direct N-body simulations," Journal of Computational and Applied Mathematics, 1999, Vol. 109, Issues 1-2, pp. 407-432.

[24] Y. Tang, K. S. Perumalla, R. Fujimoto, H. Karimabadi, J. Driscoll and Y. Omelchenko, "Optimistic Simulations of Physical Systems Using Reverse Computation," SIMULATION, 2006, vol. 82, issue 1, pp. 61-73.

[25] S. P. Zwart, S. McMillan, D. Groen, A. Gualandris, M. Sipior, W. Vermin, "A parallel gravitational N-body kernel," New Astronomy, 2008, Vol. 13 Issue 5, pp. 285-295.